Applied Research Laboratory

Technical Report

PARTICLE TRACE ANIMATIONS FOR 2-D, STEADY STATE, MULTIPHASE, RECIRCULATING FLOW FIELDS IN A CLOSED BATH COMBUSTOR OF LIQUID METAL FUELS

by

T. T. Blackmon
T. F. Miller

PENNSTATE

1855

The Pennsylvania State University
APPLIED RESEARCH LABORATORY
P.O. Box 30
State College, PA 16804

PARTICLE TRACE ANIMATIONS FOR 2-D, STEADY STATE, MULTIPHASE, RECIRCULATING FLOW FIELDS IN A CLOSED BATH COMBUSTOR OF LIQUID METAL FUELS

by

T. T. Blackmon
T. F. Miller

Technical Report No. TR 96-005
June 1996

Supported by:
Space and Naval Warfare Systems Command

L.R. Hettche, Director
Applied Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** ~~May 1999~~ 6/96	**3. REPORT TYPE AND DATES COVERED** Undergraduate Honors Thesis

4. TITLE AND SUBTITLE Particle Trace Animations For 2-D, Steady State, Multiphase, Recirculating Flow Fileds in a Closed Bath Combustor of Liquid Metal Fuels

5. FUNDING NUMBERS

1999

6. AUTHOR(S)

T. T. Blackmon
T. F. Miller

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Applied Research Laboratory
The Pennsylvania State University
P. O. Box 30
State College, PA 16804

8. PERFORMING ORGANIZATION REPORT NUMBER

TR#96-005

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Space and Naval Warfare Systems Command
Code OOL
2451 Crystal Drive
Arlington, VA 22202

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

Includes Video Tape: Particle Trace Animations

12a. DISTRIBUTION/AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A CFD post processing technique was designed which employs computer graphics to generate particle trace animations for 2-D, steady state, multiphase, closed, recirculating flow fields. A particle trace algorithm, which uses a Huen predictor/corrector time stepping method, a Lagrange shape function interpolation method, and a special stopping criteria for recirculating flow fields, calculates the path a Lagrangian particle in an Eulerian flow field. Then, computer graphics techniques are used to represent and animate fluid particles along their calculated paths. This technique was developed for a single phase flow field, then extended to a multiphase flow field which models a closed bath combustion of a liquid metal fuel. Capabilities were developed to plot scalar contours of flow field quantities as background images for the particle trace animations. Scalar contours of x-ray attenuation levels were then plotted for a multiphase flow field to simulate x-ray radiographs of closed combustion of liquid metal fuels. Volume fraction data of the phases was also used to plot background color contours and to control the individual color intensities of multiple color fluid particles representing the various phases.

14. SUBJECT TERMS

Scientific Visualization
Flow Visualization
CFD

15. NUMBER OF PAGES 141

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	**18. SECURITY CLASSIFICATION OF THIS PAGE** UNCLASSIFIED	**19. SECURITY CLASSIFICATION OF ABSTRACT** UNCLASSIFIED	**20. LIMITATION OF ABSTRACT**

Abstract

A CFD post processing technique was designed which employs computer graphics to generate particle trace animations for 2-D, steady state, multiphase, closed, recirculating flow fields. A particle trace algorithm, which uses a Huen predictor/corrector time stepping method, a Lagrange shape function interpolation method, and a special stopping criteria for recirculating flow fields, calculates the path a Lagrangian particle in an Eulerian flow field. Then, computer graphics techniques are used to represent and animate fluid particles along their calculated paths. This technique was developed for a single phase flow field, then extended to a multiphase flow field which models a closed bath combustion of a liquid metal fuel. Capabilities were developed to plot scalar contours of flow field quantities as background images for the particle trace animations. Scalar contours of x-ray attenuation levels were then plotted for a multiphase flow field to simulate x-ray radiographs of closed combustion of liquid metal fuels. Volume fraction data of the phases was also used to plot background color contours and to control the individual color intensities of multiple color fluid particles representing the various phases.

Table of Contents

List of Figures........ i

Chapter 1: Introduction........ 1

Chapter 2: Background
- Flow Patterns and Visualization........ 3
- Experimental Flow Visualization........ 7
- Numerical Flow Visualization........ 14

Chapter 2: Particle Path Calculations........ 22
- Eulerian vs. Lagrangian Descriptions........ 22
- What Path?........ 23
- Mechanics of Path Calculations........ 24
- Initial Conditions........ 25
- Time Step Interval........ 26
- Time Stepping Methods........ 27
- Interpolation Methods........ 37
- Stopping Criteria........ 49
- Boundary Crossing........ 55
- Particle Trace Algorithm........ 56

Chapter 3: Mechanics of Computer Graphics........ 57
- Drawing Graphical Primitives........ 57
- Higher Order Drawing Through the Use of NURBS........ 59
- Graphical Objects and Double Buffering........ 62
- Controlling Colors and Color Mode........ 67
- Shading........ 69
- Blending........ 70

Chapter 4: Applied Examples........ 72
- Mint Flow Field........ 73
- Radiograph Simulations........ 78

Chapter 6: Conclusion........ 85

Appendix A: Linear Equation Solver........ 89
Appendix B: Source code for Apt1........ 92
Appendix C: Modified Subroutines for other Apt's........ 106

List of Figures

		pg.
Fig. 2.1	Velocity vector components.	4
Fig. 2.2	Deflection of a light ray in a variable density media	12
Fig. 2.3	Local flow rate vectors for a multiphase flow field	20
Fig. 2.4	Volume fraction contours for a multiphase flow field	21
Fig. 3.1	Time stepping methods comparison	34
Fig. 3.2	Particle location with respect to nodal network	37
Fig. 3.3	Interpolation methods comparison	46
Fig. 3.4	Master element in natural coordinates	41
Fig. 3.5	Stopping criteria for recirculating flows	51
Fig. 4.1	Control polygon for B-spline	60
Fig. 4.2	Graphical objects to store particle traces	65
Fig. 5.1	Digitized x-ray radiograph of a closed bath combustor at 60% utilization	77
Fig. 5.2	Contours of numerically calculated x-ray attenuation levels for the x-ray flow field	78
Fig. 5.3	Color contour background images for x-ray particle trace animations	80

Chapter 1: Introduction

The traditional post processing technique for displaying velocity vector field output from Computational Fluid Dynamics (CFD) is to simply plot vectors representing the magnitude and direction of the velocity field at the nodal locations. Common alternative methods for numerical flow visualization of velocity vector fields include mathematically calculating the streamlines or path lines (particle traces) of the flow field and then either plotting static images of these flow patterns or animating fluid particles along the flow patterns. The objective of this project was to develop a CFD post processing technique for generating animation sequences of 2-D, steady state, multiphase, closed, recirculating flow fields using particle systems.

Although particle animation programs have been previously developed, this project is unique in that the emphasis was placed on multiphase, closed, recirculating flows. Special consideration was given to a common problem that occurs with the stopping criteria for the calculation of a particle trace for a recirculating flow field. By the very definition of a steady state, recirculating flow, the path of a fluid particle should be a closed loop. However, due to numerical inaccuracies in the CFD solution and the particle trace calculation, spiraling and overshoot of the calculated path arise if a particle trace does not return to its exact initial location. Therefore, a stopping criteria was developed for the particle trace calculations which helps eliminate the path spiraling and overshoot effects particular to recirculating flow fields.

In addition, animating multiphase flows presents another challenge in visually distinguishing between the several components of the flow field. Although CFD solution files include information related to the volume fractions of the particular phases at the nodal locations, methods must be developed to visually convey this information during the particle trace animation. Several techniques were explored to address this problem. One technique was based on an existing experimental method for visualizing multiphase flows -- X-ray radiography. Another technique uses the volume fraction data to properly texture the animated fluid particles with color and transparency.

This project is also unique in the approach taken to animate the fluid particles along the calculated paths. A method to animate the fluid particles was developed which separates the particle trace calculations from the animation sequences and stores the calculated path information as a series of images rather than a numerical data base. This drastically reduced the computational requirements in CPU speed and memory that limit other methods of animating particle traces. Furthermore, the computationally efficient method that was developed enables the animation of multiple fluid particles along numerous particle traces, while preserving smooth motion, so that a dynamic picture of the entire flow field pattern may be visualized.

Chapter 2: Background

Generating animation sequences of single phase and multiphase flow fields using CFD output requires an understanding of the concepts of streamlines, pathlines, streaklines, and timelines. Traditional methods of experimental and numerical flow visualization should also be investigated. This background will provide a basis for developing the techniques used in this work. The following sections provide the background of flow visualization that was researched for the project.

Flow Patterns and Visualization

Streamlines

A streamline is defined as a line tangent to the velocity vector at a given instant. [1] Streamlines may be found using other flow vectors as well since they are defined by multiplying the velocity vector by a scalar and thus do not affect the vector direction. Streamlines can be easily calculated mathematically from the velocity field. For every vector **dr** which is tangent to **V**, the respective components must be in exact proportion, see Figure 2.1. Mathematically...

$$\frac{dx}{u} = \frac{dy}{v} = \frac{dz}{w} = \frac{d\mathbf{r}}{\mathbf{V}} \tag{2.1}$$

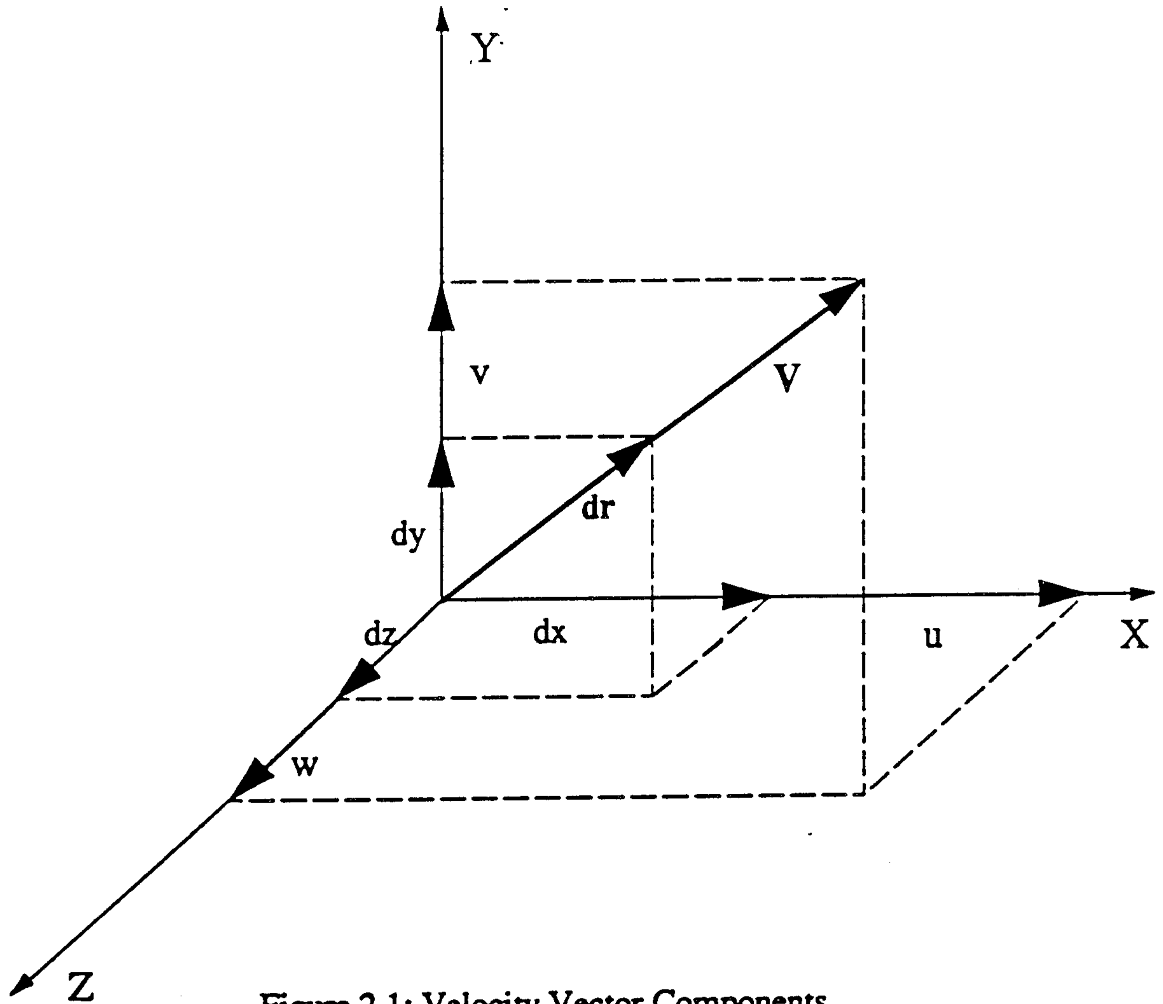


Figure 2.1: Velocity Vector Components

Thus, if the velocity components u, v, & w are known functions of time and space, Eqn. 2.1 can be integrated to find the streamline passing through (x_o, y_o, z_o, t_o). However, the calculations may be laborious and tedious. Alternatively, the calculations may be simplified by introducing a variable ds and setting it equal to the ratios given in Eqn. 2.1 such that...

$$\frac{dx}{ds} = u\,;\ \frac{dy}{ds} = v\,;\ \frac{dz}{ds} = w \tag{2.2}$$

Now, Eqn. 2.2 can be integrated with respect to s, holding t constant, with the initial condition (x_o,y_o,z_o,t_o). Then s may be eliminated to obtain f(x,y,z,t) which represents the streamline.

Pathlines

A pathline is defined as the actual path traversed by a given fluid particle. [1] However, unlike the streamlines, the pathlines calculated for the velocity field will not coincide with the pathlines calculated for other flow vectors. For a velocity field, a pathline may be calculated through the integration of the relationship between velocity and displacement...

$$dx = u * dt \; ; \; dy = v * dt \; ; \; dz = w * dt \tag{2.3}$$

Eqn. 2.3 may be integrated with respect to time using the initial condition (x_o,y_o,z_o,t_o). Then, time may be eliminated to yield the pathline function f(x,y,z).

Alternatively, a pathline may be numerically computed by choosing a set of initial conditions and time stepping a fluid particle through the flow field. This typically requires selecting an appropriate time step interval, small enough to accurately compute the pathline, interpolating the fluid particle velocity components from the known nodal velocities, and establishing the appropriate stopping criteria for the path calculation.

Streaklines

A streakline is defined as the locus of fluid particles which have earlier passed through a prescribed point. [1] Streaklines may also be calculated mathematically through the integration of the relationship between velocity and displacement. Again, Eqn. 2.3 may be integrated with respect to time. However, retaining time as a parameter, integration constants may be found which cause the pathlines to pass through (x_o, y_o, z_o) for a sequence of times $C < t$. Then, c may be eliminated from the result to obtain the streakline function $f(x,y,z)$.

Timelines

A timeline is defined as the set of fluid particles that form a line at a given instant. [1] A timeline can be calculated mathematically by first determining the pathlines for a column of initial locations. Then, a timeline is found by joining the endpoints of the pathlines at a given instant. Although this does not directly yield an equation representing the timeline, a graphical image of the timeline may be obtained and an equation of the timeline may be generated with an appropriate curve fit.

How are these flow patterns related ?

Streamlines, pathlines, and streaklines converge onto the same curve for a steady state flow field. However, these lines are not identical for unsteady flows. Reference 2 illustrates this principle using experimental flow visualization of water over an oscillating plate. This work focused on 2-D, steady state, recirculating flow fields. Therefore, the calculated particle traces represent the streamlines and streaklines in addition to the pathlines of the flow field. In addition, the calculated particle traces should return to the initial location to ensure the closure of streamlines in a recirculating flow field.

Experimental Flow Visualization

Experimental flow visualization may be classified as intrusive or non-intrusive. Intrusive experimental flow visualization involves methods that in some way directly interact with the flow field. However, the disturbing effects of most intrusive techniques can be minimized to be insignificant. Non-intrusive experimental flow visualization involves methods that do not interfere with the flow field.

There are a wide range of both intrusive and non-intrusive experimental flow visualization techniques. The more interested reader may refer to the literature for detailed explanations covering a variety of experimental flow visualization techniques. [5-11] The following intrusive and non-intrusive flow visualization techniques were determined to be relevant for this project.

Intrusive:

1) Foreign Material Additions
2) Electrolytic Techniques

Non-intrusive:

1) Optical Techniques

2) Radiography

Intrusive Experimental Flow Visualization

Foreign Material Additions

A widely used intrusive experimental flow visualization technique involves the addition of foreign material into a gaseous or liquid fluid flow. Thus, the observer actually views the motion of the added material, not the motion of the fluid. However, the difference between the fluid motion and the particle motion can be made negligible by using a foreign material whose density is the same magnitude as the fluid density. [5] In doing so, the particle motion accurately represents the direction and magnitude of the fluid velocity.

Foreign particle additions usually involve the release of dye, smoke, or solid particles at discrete points in the experimental flow field. Foreign particle additions provide an excellent and easy means for visualizing fluid motion. When such foreign material is released into the flow field, its motion is influenced by the mean flow velocity. Thus, different classes of flow patterns may be visualized depending on how the foreign material is introduced and released into the flow.

Streamlines can be experimentally visualized by releasing numerous small particles throughout the flow field. Then, a photograph is taken with a known exposure time so that each particle appears as a streak. This directly yields the magnitude and direction of the velocity at all particle locations over an extremely short interval of time. The streamlines may then be superimposed on the photograph as curves which are tangential to these particle streaks.

Streaklines can be experimentally visualized through the continuous release of dye, smoke, or particles from a specified location in the flow field. Because streaklines are defined as the locus of fluid particles which have earlier passed through a prescribed point, it is important that the selected location from which the foreign material is released be fixed in time and space.

Pathlines may be generated experimentally by the time exposure of a single marked particle moving through the flow. Release of a single particle into the flow field can provide insight into the flow near a critical area such as a boundary layer. Alternatively, the release of numerous particles into the flow provides an excellent means for gaining an understanding of the flow field.

A pair of films which demonstrate the use of foreign particle additions as a tool for experimental flow visualization are given as references [2-3]. The use of foreign particle additions is also graphically illustrated through photography in reference [4].

Electrolytic Techniques

The primary electrolytic technique involves the release of hydrogen bubbles into an experimental flow field. This method is similar to the release of dye, smoke, or particles in that the motion of small hydrogen bubbles is used represent the fluid motion. However, hydrogen bubble injection is based on the process of electrolysis. If the fluid of interest is an electrolytic conductor, gas bubbles may be generated within the flow field through electrolysis. An excellent example is the generation of hydrogen and oxygen bubbles in an aqueous solution (water is an electrolytic conductor).

Two electrodes are placed in the flow field and a DC current is applied across the electrodes. This causes the formation of oxygen bubbles at the anode and hydrogen bubbles at the cathode. Because hydrogen bubbles are lighter and smaller than the oxygen bubbles, the hydrogen bubbles have less of an effect on the flow. Therefore, only the hydrogen bubbles are used as the tracers and the oxygen bubbles can be eliminated by placing the anode outside the region of interest. The primary advantage of hydrogen bubble injection is that the rate of release of the hydrogen bubbles can be easily controlled. [5] Also, normal tap water may be used as the fluid since it is an electrolytic conductor.

Hydrogen bubble injection is most often used for the experimental visualization of timelines and velocity profiles. By using a thin wire as the cathode, pulses of hydrogen particles may be released at a given instant. The wire can thus be stretched across the flow field in any feasible fashion to produce appropriate timelines and velocity profiles. This technique is an excellent way to visualize the velocity differences between the top and bottom of a wing section or to visualize the velocity profile near a wall.

Non-intrusive Experimental Flow Visualization

Optical Techniques

Experimental flow visualization methods which are non-intrusive often involve an optical technique applied to variable density fluids. Because of the variation in fluid density, intrusive techniques such as the addition of foreign material cannot be used. However, it is possible to use the principal that the optical index of a fluid is a function of fluid density for experimental flow visualization.

In optical techniques, rays of light are passed through a variable density flow field. Each ray of light is thus disturbed by the variation of the optical index of refraction caused by the variation in fluid density. This has two simultaneous effects on the light rays.

1) The light rays are deflected from their original direction.
2) The light rays experience a phase shift owing to different optical path lengths

These effects are the basis for experimental flow visualization which use optical techniques in variable density flows.

Figure 2.2 simulates the effect of variation in gas density (and subsequent variation in optical index of refraction) on rays of light which pass through the flow field.

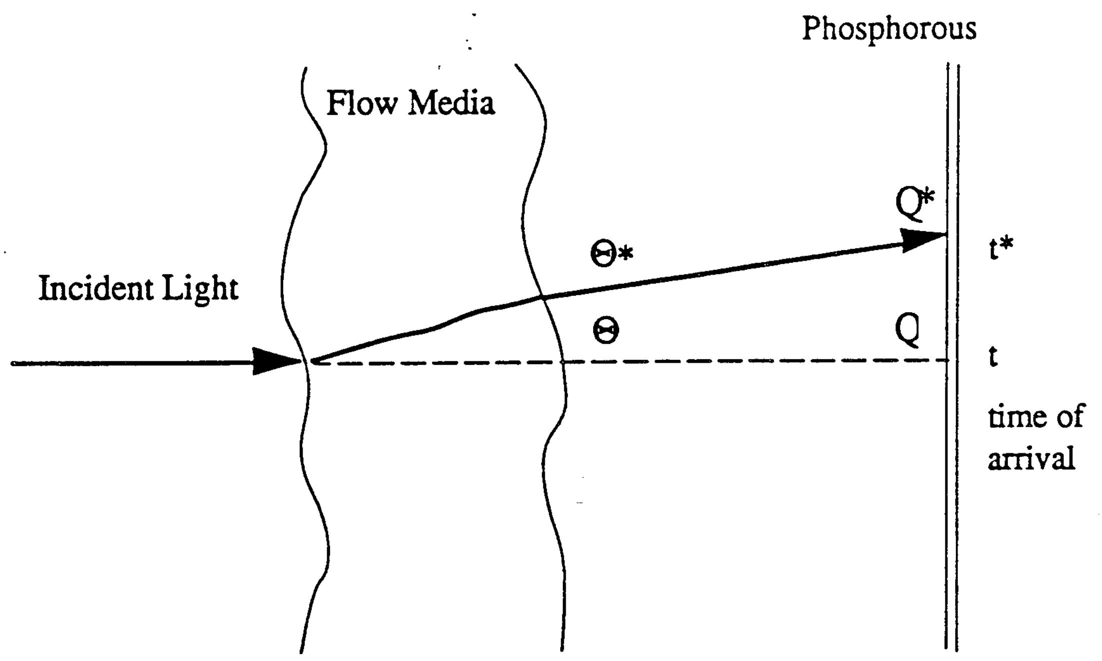


Figure 2.2: Deflection of a light ray in a variable density flow media

By assuming that the index of refraction is a function of space only and not time (stationary flow), the variation in gas density effects the following quantities.

1) Displacement QQ*
2) Angular deflection $\Theta - \Theta^*$
3) Phase shift $\omega - \omega^*$

Each of these quantities are used as the basis for an optical technique in variable density flows. [5] The shadowgraph technique is based on the displacement QQ*. The schlieren technique is based on the angular deflection $\Theta - \Theta^*$. The interferometer technique is based on the phase shift $\omega - \omega^*$, due to the difference in path length.

Radiography

Radiography is a non-intrusive experimental flow visualization technique which is extremely useful for multi-component flows and for flows with limited optical access. The underlying principle of radiography is to form an image through the attenuation of x-rays or neutrons by different media. The attenuation, A, of a multi-media system to incoming x-rays or neutrons is given by...

$$A = 1.0 - \sum_{j=1}^{n} (K(j)*x(j)) \tag{2.4}$$

where....

n = total number of different media present

K(j) = attenuation coefficient or cross-section of media j

x(j) = distance the x-ray or neutron must travel through media j

If x-rays or neutrons are targeted at the flow field of interest, the variations in attenuation levels will appear as bright and dark regions on a phosphor screen placed opposite the incoming x-rays or neutrons. From the image formed through the attenuation of x-rays or neutrons, a spaced averaged approximation of the flow composition may be obtained through knowledge of the attenuation levels of the particular media in a multi-phase flow field. Furthermore, real-time radiography (RTR), or continuous imaging of a multi-component flow field with the aid of video processing equipment, can provide a dynamic flow pattern visualization of the most attenuating species. RTR has been successfully applied by

Parnell, et al [12-13] to study the closed combustion of liquid metal fuels. Here, x-ray radiography provides a 2-D view of a rectangular, 3-D, multiphase flow field.

Numerical Flow Visualization

The computer provides a powerful tool for the study of fluid flow phenomena. Computational Fluid Dynamics (CFD) has become an integral part of fluid mechanics research and design, joining experimental and analytical methods as the primary basis of study. Because of its relative speed and reduced cost in comparison to experimental methods, CFD is widely used as a preliminary research and design tool. Then, in the later stages of a project, experiments can be conducted to verify final designs and results.

The output generated by CFD, however, can be quite large, generally consisting of data files of flow quantities at nodal locations as well as the nodal coordinates. Thus, the extensive use of computer graphics to post process and display CFD output has been developed as a primary method for understanding the results of CFD. Furthermore, the fluid investigator is usually concerned with more than just flow quantities at specific locations. Trends in fluid properties and critical points in the flow field, which provide considerable insight into the flow physics, can be easily visualized and perceived with the aid of computer graphics. Computer software packages for numerical flow visualization are commercially available and most CFD packages include their own post processing routines as well as the grid generation (geometric modeling) and solving routines. PLOT3D [14], developed by Pieter Buning at NASA Ames Research Center and FAST [15] are excellent examples of CFD post processing software packages which run on a Silicon Graphics IRIS-4D workstation. Buning has also published numerous papers on flow visualization in CFD and the use of computer graphics for numerical flow visualization. [16-18] Common numerical flow visualization techniques include vector plots, contour plots, and particle traces. More recent developments in numerical flow visualization include vector field topology and animated particle traces.

Vectors

Plotting vectors at nodal coordinates to represent the magnitude and direction of flow quantities such as velocity and flow rate is a conventional and widely used numerical flow visualization technique. Figure 2.3 is a computer graphics display of a velocity vector plot for a 3-D, multiphase, CFD flow field. [19] The image was generated using PLOT-3D software on an IRIS-4D GT workstation. Computer software for plotting vectors allows the user to import nodal coordinates and vector components at the nodes. Then the user can scale, color and shade the vectors to obtain a picture which visually represents the vector field.

However, because of the spatial complexity and variation in relative magnitudes of vector fields, plotting vectors can sometimes result in a cluttered display which is difficult to comprehend. This is particularly true for large, complex, 3-D vector fields, as in Figure 2.3. In these cases, particular regions of interest in the flow field or

2-D cross-sections may be displayed to reduce the data set and simplify the picture. However, this limits the amount of information that the user can obtain from the vector field. Nevertheless, vector plots remains a primary technique for numerical flow visualization.

Contours

Contours of scalar quantities such as pressure, density, and temperature may be represented using lines, colors or surface height variation over the flow field. Typical CFD post processing software allows the user to input a scalar data file and define a scale, or range, of colors or surface height to display the variation in the scalar quantities over a region of interest. In addition to providing considerable insight through a concise picture into the variations in a single flow variable, plotting contours also reveals observable trends and relationships between flow field variables. Figure 2.4 is a computer graphics display of volume fraction contour plots using iso-lines for the three components of a multiphase flow field using PLOT-3D software on an IRIS-4D GT workstation. [19]

Vector Field Topology

Vector field topology has recently been developed as a tool for visualizing the key aspects of a vector field. [20-21] The vector field topology consists of the critical points in a vector field, where the magnitude of the vector goes to zero, and the integral curves and surfaces that connect the critical points. For a CFD velocity vector field, the critical points may be sinks, sources, attracting and repelling spirals, and saddles. The integral curves are the particle traces, streamlines, and stream surfaces which can be generated from the integral curves and surfaces by choosing appropriate initial conditions. Thus, a topological map of a vector field showing the critical points and the integral curves and surfaces can provide an uncluttered picture describing the important features of a large, complex, 3-D vector field. Vector field topological capabilties exist in the FAST software package.

Particle Traces

Particle traces are simply the numerically calculated pathlines of a CFD flow field. By specifying a set of initial conditions, which include the initial location and initial velocity of a fluid particle, and a time step size, the path of a fluid particle may be calculated. The calculation of particle traces involves using a time stepping method to march a fluid particle in time, an interpolation method to determine fluid particle quantities from given nodal quantities, and a stopping criteria to determine the end of a path calculation.

Particle traces may be easily visualized using computer graphics. Conventionally, a particle trace is represented as a static image of a curve or series of connected line segments which pass through the calculated points of a particle path. Therefore, to generate a static image of a particle trace requires specification of initial conditions and time step size, calculating the particle trace, then displaying the particle trace on an appropriate coordinate system with respect to the flow field geometry.

More recent efforts have gone towards animating particle traces. By providing a dynamic picture of the fluid motion through the solved velocity field, particle trace animations allow a viewer to readily observe changes in flow direction and speed, easily recognize flow patterns, and obtain qualitative insight into the general flow field characteristics. By animating a relatively few number of particle traces, an observer can track individual fluid particles along their calculated paths and obtain an understanding of localized behavior of the flow field. When a very large number of particle traces are used in the animation, an understanding of the overall flow field is obtained.

However, animating particle traces is a technical challenge in terms of computer speed and memory. Presently, to animate a fluid particle in real-time, as the particle trace is calculated, and still observe smooth motion, either requires the speed of a CRAY super computer or limits the particle trace animation to a very small number of fluid particles. [22] This may be overcome by filming stills of the particle traces at individual time steps then playing the stills back with a video recorder. However, this method is cumbersome and time consuming. Another option, which involves first calculating the particle trace and storing the path information as a data file, then using the path data for the animation, reduces the extensive CPU speed requirement of real-time particle trace animations; however, the memory required to store the path calculations becomes too large if numerous particle traces are to be animated. Therefore, creative approaches must be developed within the constraints of computer graphics capabilities and computer speed and memory if particle trace animations are to be generated using a modern workstation. Examples of work with particle traces and particle systems is given in references [23-24].

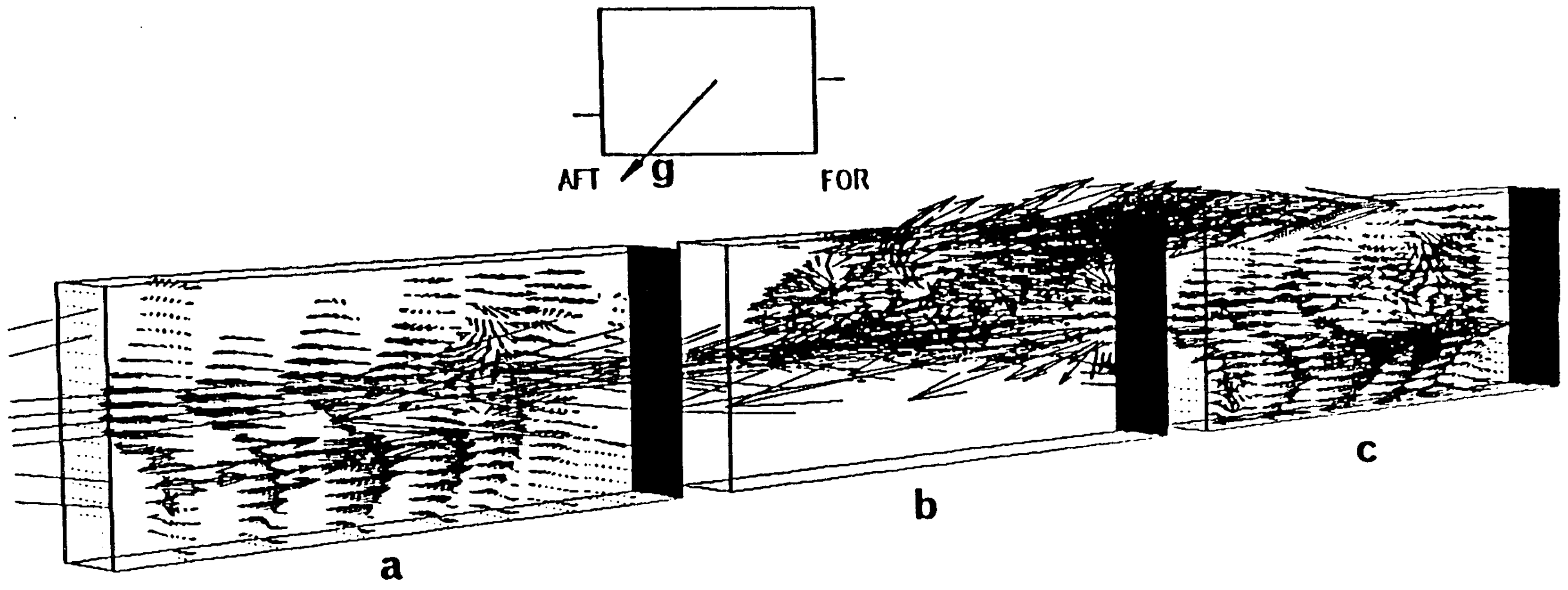


Fig. 2.3 Local flow-rate vectors of a) fuel, b) vapor, and c) product. Operating conditions are those described for Fig. 3.

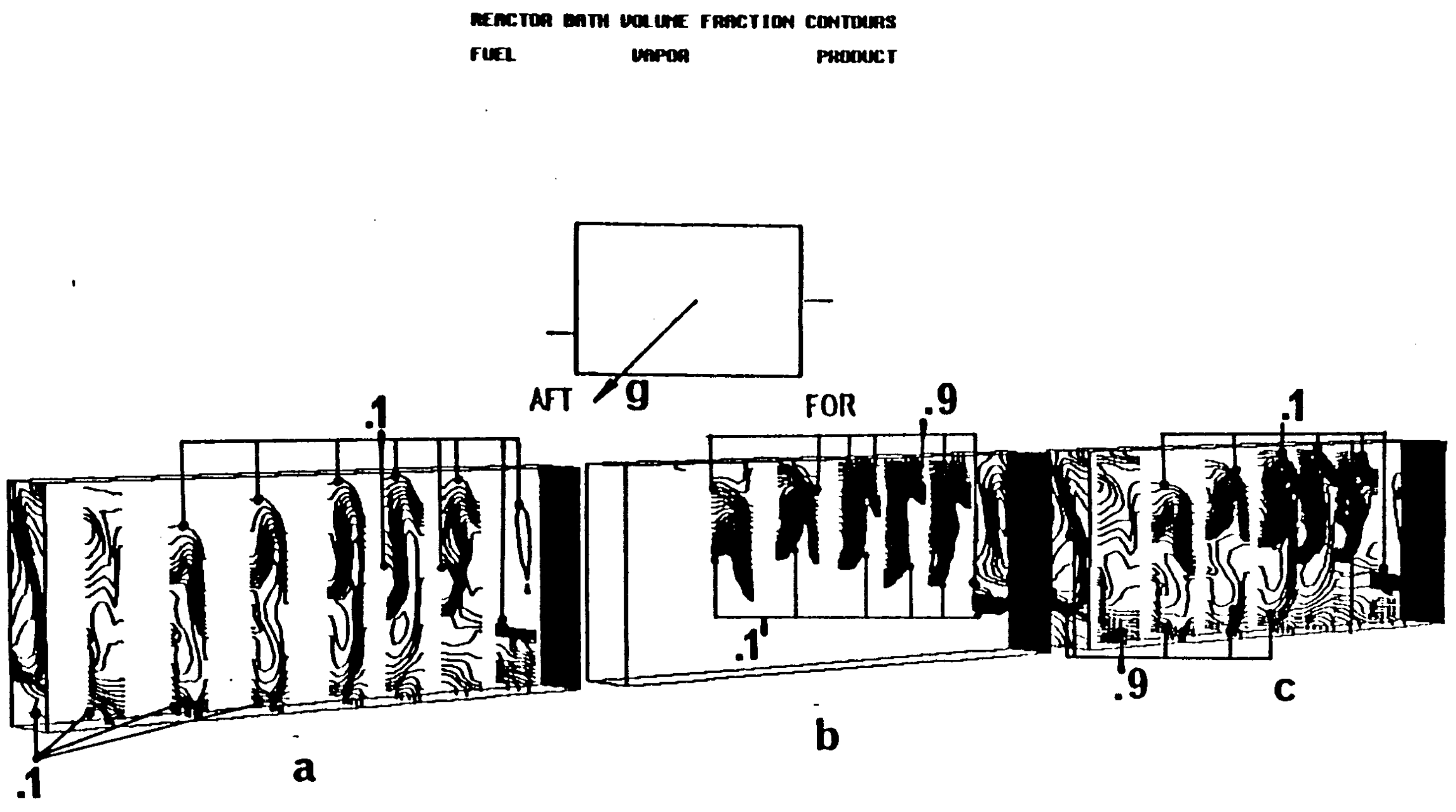


Fig 2.4 Volume fraction contours of a) fuel, b) vapor, and c) product. Operating conditions are those described for Fig. 3.

Chapter 3: Particle Path Calculations

Eulerian vs. Lagrangian Descriptions in Fluid Mechanics

There are two distinct descriptions used in fluid mechanics -- the Eulerian and the Lagrangian. [1] The Eulerian method concerns the flux of fluid or particles through a specific control volume. Flow field values such as the pressure, $p(x,y,z,t)$, temperature, $T(x,y,z,t)$, density $\rho(x,y,z,t)$, and velocity $V(x,y,z,t)$ are computed as functions of space and time. Thus, the Eulerian method emphasizes the space-time distribution of fluid properties.

Alternatively, the Lagrangian method is concerned with individual particles moving through the flow field. In the Lagrangian method, the control volume is usually the boundary of the particle under consideration. The positions $x(t)$, $y(t)$, & $z(t)$ as well as the fluid properties of pressure, $p(t)$, temperature, $T(t)$, density, $\rho(t)$, etc.... are computed as a function of time for an individual fluid particle. Thus, the Lagrangian method emphasizes the changes of fluid properties for an individual fluid particles as it moves through the flow field.

The Eulerian method is more widely used in fluid mechanics. Scientists and engineers are usually concerned with global flow quantities, that is the space-time distribution of fluid properties, and the Eulerian method yields these results. However, the Lagrangian method may be more suitable for some sharply bounded flow problems such as the behavior of an isolated fluid droplet descending through the atmosphere. Also, interparticle collisions and interactions can adequately describe a continuum flow field when a statistically significantly number are calculated using Lagrangian techniques.

Because the Eulerian method is more widely applicable in fluid flow analysis, the data sets used as input for CFD post processing are usually obtained from an Eulerian flow field. However, to animate particles in an Eulerian flow field requires determination of individual particle paths, which is a Lagrangian technique. Therefore, the goal is to develop a method for calculating the path of a Lagrangian particle in an Eulerian flow field.

What Path ?

To calculate the path of a Lagrangian particle in an Eulerian flow field, the physical significance of the particle path must be determined. By definition, the flow pattern obtained from tracing the path of a fluid particle is a pathline. Thus, the experimental analog would be a foreign material addition to produce pathlines. However, by conveniently assuming that the particle's velocity is exactly equal to the fluid velocity at the location of the particle, the particle in no way effects the flow quantities. Thus, the numerical technique is truly non-intrusive, unlike the experimental analog.

Furthermore, if the flow field is steady, then the particle path also represents a streamline and a streakline of the flow field. Even if the flow is unsteady, streaklines and timelines may be generated by controlling the release of the particles into the flow field. A streakline may be observed by the release of numerous particles from a fixed location over an interval of time. Similarly, a timeline may be observed by the simultaneous release of a column of particles at a given instance.

The extension to multiphase flows is not as obvious. The question arises as to which flow vectors should be used to "move" a fluid particle during a path calculation? A typical CFD output file for a multiphase flow consists of nodal velocity components for each phase as well as the volume fractions of each phase at the nodes. Therefore, various combinations of velocities, or flow vectors, may be used in the path calculations.

As stated previously, when a foreign material is released into a flow field, its motion is influenced by the mean flow velocity. Thus, to remain consistent with the experimental analog of foreign material addition, the mean flow velocity of the multiphase flow field should be used as the particle velocity in the path calculations. The mean flow velocity for a multiphase flow field is the volume fraction average of the phase velocity components. Mathematically...

$$V_m = \Sigma \Theta_i {}^* V_i \qquad (3.1)$$

where...

V_m = mean flow velocity

Θ_i = volume fraction of phase i such that $\Sigma\Theta_i = 1$

V_i = velocity of phase i

However, it is possible to use other combinations of flow vectors for the particle path calculations of a multiphase flow field. For example, separate particle traces may be calculated for each phase by using the velocity components of a particular phase to move the Lagrangian fluid particle. Then, the particle traces for each particular phase may be distinguished using colors. Also, since experiemntal x-ray radiographs show the movement of the most attenuating component of the flow field, attenuation data may be plotted as color contours for the particle trace animations and the velocity components of the most attenuating species may be used to calculate the particle traces.

Mechanics of Path Calculations

Calculating the path of a Lagrangian particle in an Eulerian flow field involves the following procedures...

i) Establishing the initial conditions and an appropriate time step interval.

ii) Time stepping the particle through the flow field.

iii) Interpolating particle quantities (namely velocity components) from the known nodal values.

iv) Stopping criteria for the calculation of a particle's path.

Initial Conditions and Time Step Interval

Establishing the initial conditions of a Lagrangian particle involves the determination of the initial location and initial velocity of the particle. The initial velocity of the fluid particle is easily calculated by equating it to the fluid velocity at the initial location. Thus, establishing initial conditions of a Lagrangian particle for a path calculation reduces to only determining the initial location. Two methods have been developed in this work and implemented for determining the initial location of a Lagrangian fluid particle.

One method allows the user to select the initial location interactively. This method is suitable for observing a select number of pathlines at specific regions of interest. A second method provides a user subroutine which instructs the program to automatically establish initial locations for numerous particles. This method is more suitable for observing aggregate fluid motion because it allows numerous particles to be released at initial locations scattered throughout the flow field. This method also allows the user to generate specific flow patterns such as streaklines and timelines.

For the particle path calculations, a constant time step interval, dt, was employed because it eliminates the need to calculate a new dt for each time step. This simplification considerably reduces computational time. Also, a variable time step is unnecessary because the flow fields used as data sets are steady state. Furthermore, a constant time step facilitates the simultaneous animation of multiple particles in the flow field.

An initial time step interval was chosen which limits the fastest particle to advance no more than $\frac{1}{2}$ grid point each time step. Thus, to determine the time step interval, the maximum velocity in the flow field and the minimum distance across its control volume are calculated. Then, the time step interval, dt, is computed by dividing the minimum distance by the maximum velocity. However, the time step interval calculated in this manner is generally too small for most flow fields because the optimum dt (max dt such that the particle motion is accurately represented) is highly dependent upon the particular flow field under observation. Therefore, the program allows the user to interactively increase the initial value of dt computed using the original method, observe the effects in computational time and path accuracy, and then select a suitable time step interval for the particular flow field.

Time Stepping Methods

Various methods have been developed for time stepping Lagrangian particles in Eulerian solved flow fields. [25] For particle path calculations, the "best" time stepping method is one that achieves the desired level of accuracy while using the least amount of computational time. This section outlines several time stepping methods and compares the methods for path calculations of Lagrangian particles in a 2-D, steady state, Eulerian flow field. Each particle path was calculated using the identical CFD data file as input, the same constant time step interval, the same interpolation method, and the same initial locations.

Eulerian Forward Step

$$x(t + dt) = x(t) + u(t) * dt \tag{3.2}$$

This is the most simple of all the time stepping methods and forms the basis for the methods that follow. In the Eulerian forward step, the new particle location only depends on the velocity at the current location. Thus, only one velocity calculation is necessary and the Eulerian forward step method requires the minimum computational time. However, the computational errors involved with the Eulerian forward step accumulate rapidly, on the order of dt. Thus, the Eulerian forward step cannot accurately calculate the particle motion without significantly decreasing the time step interval, thus raising computational cost. Figure 3.1(a) shows particle traces using an Eulerian forward time stepping method for several starting locations in a 2-D, steady state, single-phase, recirculating flow field in a closed reactor.

Leapfrog

$$x(t + dt) = x(t - dt) + 2 * u(t) * dt \tag{3.3}$$

For the leapfrog time stepping method, the first time step is taken with a Huen time step to obtain an intermediate location. Then the particle is moved again from the original location to the final location with an Eulerian forward step that uses a modified velocity equal to twice the velocity at the intermediate location . After the forward step is taken, the current location becomes the intermediate location and the process is repeated. This method is similar to the Huen time stepping method in that the forward velocity field influences the particle movement. However, the leapfrog method uses twice the forward velocity (rather than the average of the forward and current velocities) to move the particle.

In doing so, the leapfrog method anticipates the particle movement, and the path accuracy is improved over the Eulerian forward time stepping method although the leapfrog method requires only one velocity calculation. Thus, the computational time of the leapfrog method is the same order of magnitude as the Eulerian forward step. However, by using such a "checkerboard" method to move a particle, the leapfrog time stepping method is extremely sensitive to error. If the path calculation begins to deviate orthogonal to the direction of motion, the particle movement can become unstable and oscillate excessively. Figure 3.1(b) shows particle traces using a Leapfrog time stepping method for several starting locations in a 2-D, steady state, single-phase, recirculating flow field in a closed reactor.

<u>Adams-Bashforth</u>

$$x(t + dt) = x(t) + (3 * u(t) - u(t - dt)) * \frac{dt}{2} \tag{3.4}$$

For the Adams-Bashforth time stepping method, the initial value of u(-dt) is calculated using an Eulerian backward time step. Then a modified velocity is calculated which is a combination of the upwind velocity and the current velocity. Thus, the upwind velocity field, not the forward velocity field, influences the particle movement.

Because the upwind velocity is simply the velocity at the previous time step, only one velocity calculation is required. Thus, the computational time for the Adams-Bashforth time stepping method is of the same order of magnitude as the Eulerian forward step. Using the upwind velocity to influence the particle movement improves the path calculations in most instances. However, the Adams-Bashforth time stepping fails to perform satisfactory where large velocity gradients exist because large velocity gradients or acceleration require anticipation of the forward velocity field. Figure 3.1(c) shows particle traces using a Adams-Bashforth time stepping method for several starting locations in a 2-D, steady state, recirculating flow field in a closed reactor.

<u>Huen (predictor/corrector)</u>

$$\begin{aligned} x_1(t+dt) &= x(t) + u(t) * dt \\ x(t+dt) &= x(t) + (u(t) + u_1(t)) * \frac{dt}{2} \end{aligned} \qquad (3.5)$$

The Huen time stepping method initially advances the particle using an Eulerian forward step. The velocity at this advanced intermediate location is then averaged with the current velocity to obtain a modified velocity. The modified velocity is then used to move the particle again from the original location to the final location using another Eulerian forward step.

By requiring two velocity calculations, the Huen time stepping method increases the required computational time by a factor of two over the Eulerian time stepping method. However, the Huen time stepping method significantly increases the accuracy of the particle path calculations compared to the Eulerian. Figure 3.1(d) shows particle traces using a Huen time stepping method for several starting locations in a 2-D, steady state, single-phase, recirculating flow field in a closed reactor.

Fourth-Order Runge-Kutta

$$x_1 = x(t) + u(t) * \frac{dt}{2}$$

$$u_2 = u(x_1),\ x_2 = x(t) + u_2 * \frac{dt}{2}$$

$$u_3 = u(x_2),\ x_3 = x(t) + u_3 * dt$$

$$u_4 = u(x_3)$$

$$x(t + dt) = x(t) + (u(t) + 2 * u_2 + 2 * u_3 + u_4) * \frac{dt}{6} \qquad (3.6)$$

The fourth-order Runge-Kutta time stepping method uses velocities at three intermediate advanced points on the forward Eulerian field as well as the current velocity to influence the movement of a particle. Thus, the fourth-order Runge-Kutta time stepping method yields a high degree of accuracy for path calculations. However, because of the increase in the number of velocity calculations required, the fourth-order Runge-Kutta increases the computational time by a factor of four over the Eulerian forward time stepping method. Figure 3.1(e) shows particle traces using a Runge-Kutta time stepping method for several starting locations in a 2-D, steady state, single-phase, recirculating flow field in a closed reactor.

Which time stepping method used?

Which time stepping method was selected as the "best" for the particle path calculations? As stated previously, for particle path calculations, the "best" time stepping method is one that achieves the desired level of accuracy while using the least amount of computational time. To generate Figures 3.1(a)-(f), a series of six starting nodal locations were selectively chosen such that the particle traces were evenly distributed throughout the flow field. Their respective particle traces were then calculated for each time stepping method and displayed as a series of line segments joining the vertices of the computed time steps. The path accuracy was compared through visual inspection and the number of timesteps were recorded to compare the computational cost of the various methods. These results are given in Table 2.1

For the time stepping methods considered, the Eulerian forward step, the leapfrog, and the Adams-Bashforth, require one velocity calculation per time step, and thus use the least amount of computational time per time step. However, the Euler forward is the least accurate and also requires a larger number of timesteps than all other methods. Also, both the leapfrog and Adams-Bashforth fail to perform satisfactorily under certain circumstances, and were thus eliminated from consideration.

The Huen predictor/corrector and the fourth-order Runge-Kutta time stepping methods both increase the amount of required computational time compared to the other methods. The Huen method requires two velocity calculations per time step while the Runge-Kutta method requires four velocity calculations per time step. Also, both significantly increase the level of accuracy in the path calculations over the Eulerian forward step. Furthermore, the relative gain in accuracy with respect to increase in computational time is greater for the Huen method rather than the Runge-Kutta method. Therefore, the Huen predictor/corrector time stepping method was chosen as most suitable for the particle path calculations.

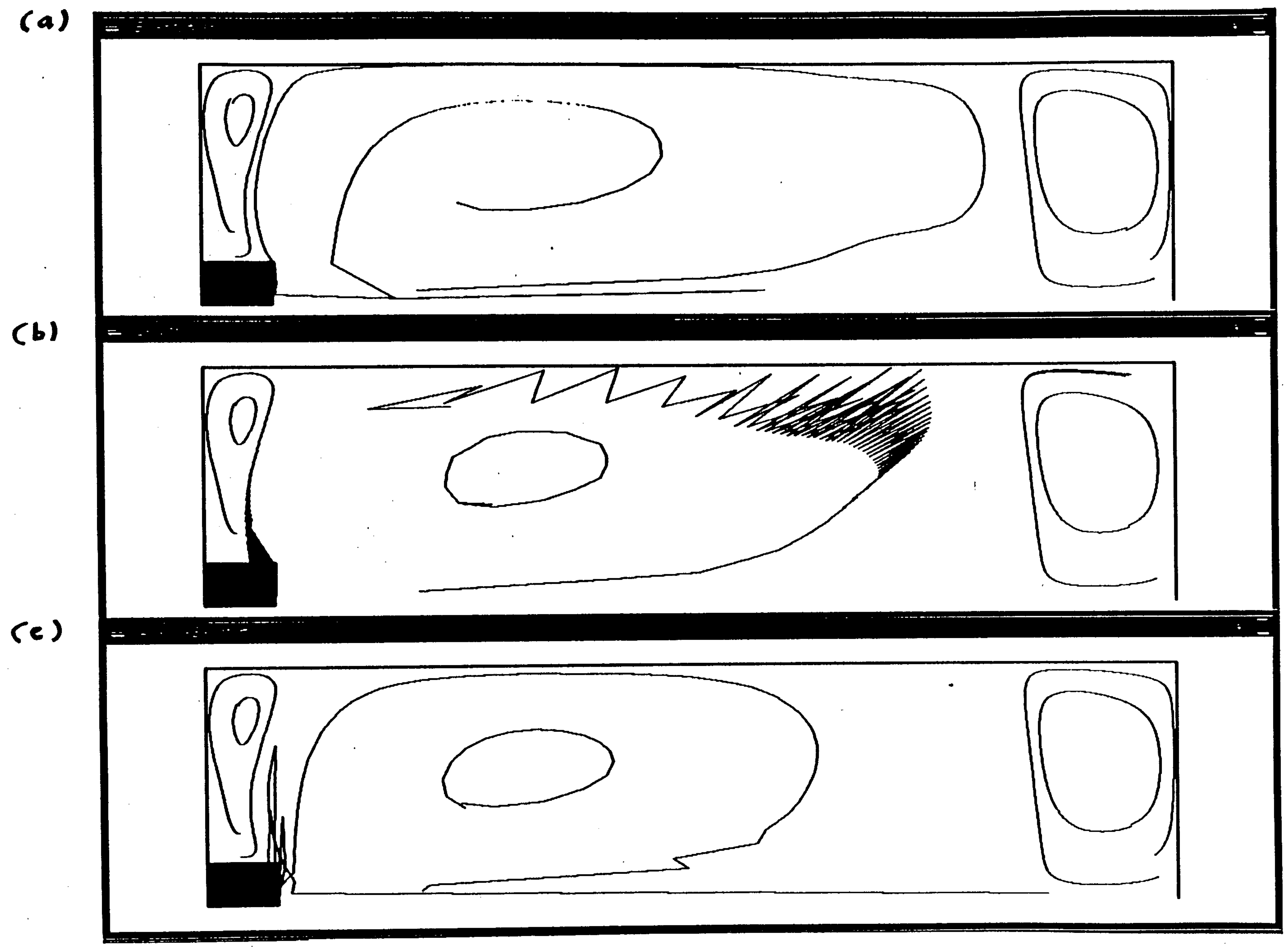
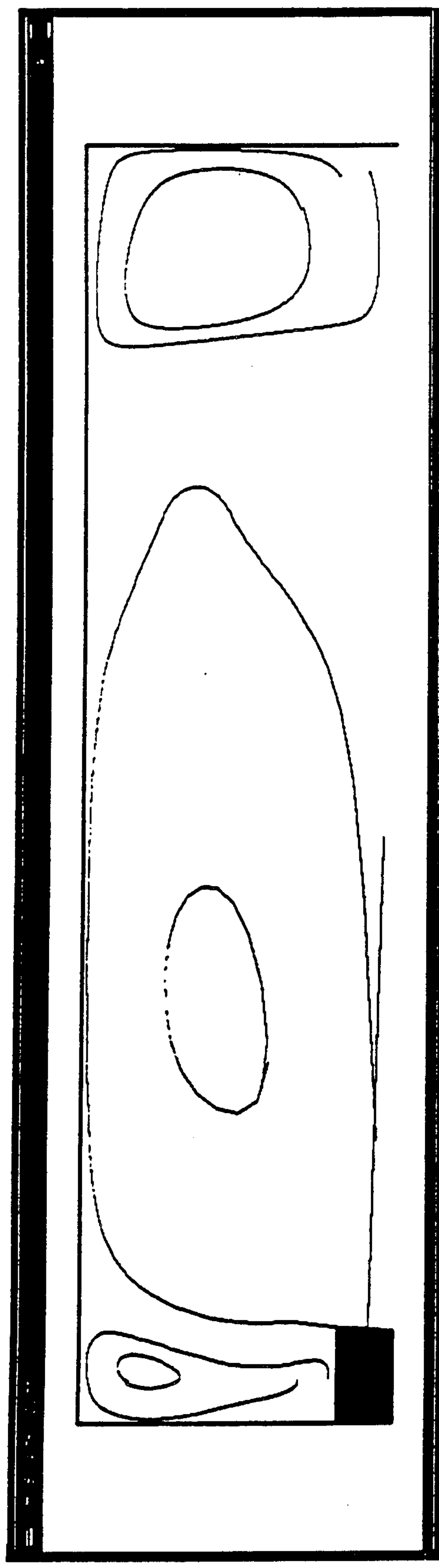


Fig. 3.1 Time stepping methods comparison: (a) Euler forward (b) Leapfrog (c) Adams-Bashforth (d) Huen predictor/corrector (e) 4th-order Runge-Kutta


(d)

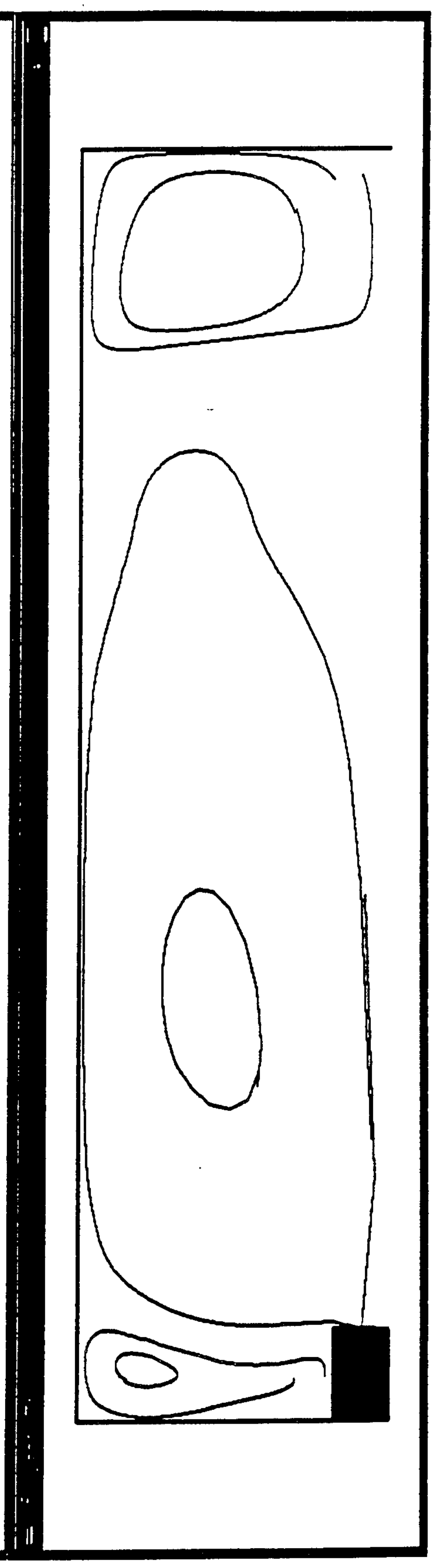

(e)

Table 3.1 : Time Stepping Methods Comparison

Data File : mint.bin
i-nodes (x-direction) : 46 (xmin,xmax) = (0.0,0.3556) m
j-nodes (y-direction) : 24 (ymin,ymax) = (0.0,0.0445) m

Linear Shape Function Interpolation

Optimum time step, dt= 7.347 e-2 s

starting location (i , j) (x , y) m	# timesteps / # velocity calculations				
	Euler	Leapfrog	Adams-B	Huen	Runge-K
(23,13) (.0936,.0188)	24/24	18/18	17/17	18/36	18/72
(40,12) (.3388,.0136)	561/561	551/551	551/551	552/1104	552/2208
(3,12) (.0118,.0136)	593/593	645/645	585/585	584/1168	584/2336
(42,6) (.3485,.0041)	857/857	532/532	843/843	843/1686	843/3372
(3,16) (.0118,..0373)	174/174	166/166	166/166	166/332	166/664
(22,5) (.0789,.0026)	185/185	97/97	68/68	130/260	138/552

Interpolation Methods

When performing a Lagrangian particle path calculation, the spatial location of the particle at time t will usually not coincide with the Eulerian nodal spatial locations. Figure 3.6 illustrates a typical situation encountered during a particle path calculation. The location of a particle, denoted by P, lies within the cell which is outlined by line segments connecting the four nodal locations, denoted by 1,2,3 & 4, that surround the particle.

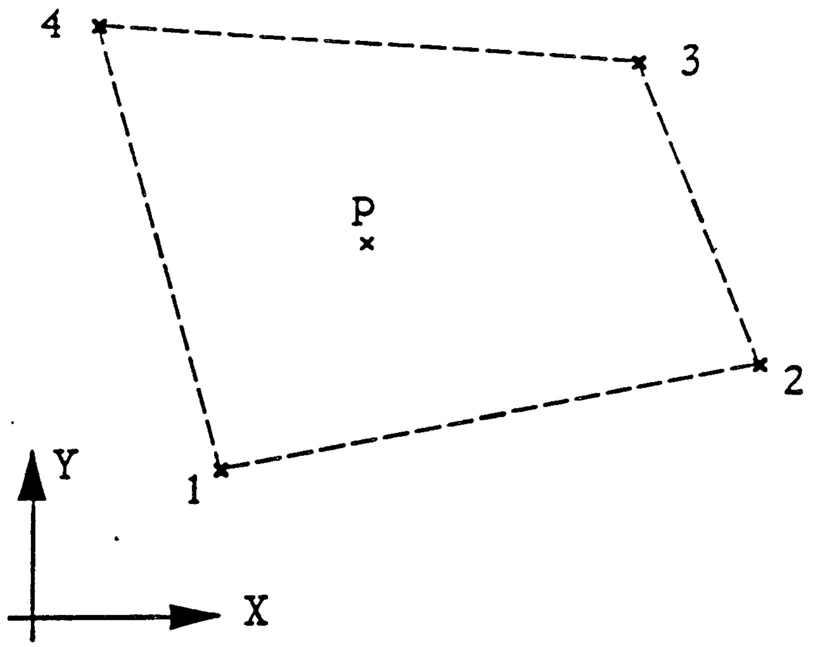


Fig. 3.2 Particle location with respect to nodal network

Thus, a particle path calculation requires a suitable method for interpolating the particle quantities, specifically the velocity components, from the nodal values based on the location of the fluid particle. This section outlines three interpolation methods for a 2-D quadrilateral grid. The interpolation methods are compared by performing identical particle path calculations and the "best" method is chosen for the computer program. Again, the "best" method is one that achieves the desired level of accuracy with the least amount of computational time.

Simple Bilinear Averaging

The easiest method for determining particle quantities as a function of nodal quantities is to perform simple bilinear averaging. This method is based on the concept of averaging the particle quantity from the nodal quantities, with the proportional influence of a nodal quantity dependent on the relative distance from the particle location to the nodal location. Figure 3.6 shows the following distances.

$$dx_1 = x_p - x_1 ; \quad dy_1 = y_p - y_1 \tag{3.7}$$

$$dx_2 = x_p - x_2 ; \quad dy_2 = y_p - y_2 \tag{3.8}$$

$$dx_3 = x_p - x_3 ; \quad dy_3 = y_p - y_3 \tag{3.9}$$

$$dx_4 = x_p - x_4 ; \quad dy_4 = y_p - y_4 \tag{3.10}$$

$$dx_{12} = dx_1 + dx_2 ; \quad dx_{34} = dx_3 + dx_4 \tag{3.11}$$

$$dy_{14} = dy_1 + dy_4 ; \quad dy_{23} = dy_2 + dy_3 \tag{3.12}$$

Using simple bilinear averaging, a particle quantity, such as the velocity component in the x-direction, may be written as...

$$u_p = \frac{dx_2{*}dy_4}{dx_{12}{*}dy_{14}}{*}u_1 + \frac{dx_1{*}dy_3}{dx_{12}{*}dy_{23}}{*}u_2 + \frac{dx_4{*}dy_2}{dx_{34}{*}dy_{23}}{*}u_3 + \frac{dx_3{*}dy_1}{dx_{34}{*}dy_{14}}{*}u_4 \quad (3.13)$$

Due to its simplicity, the simple bilinear averaging interpolation method requires little computational effort but lacks accuracy. Figure 3.3(a) shows particle traces using a simple bilinear interpolation method for several starting locations in a 2-D, steady state, single-phase, recirculating flow field in a closed reactor.

4-point Taylor Series Approximation

The 4-point Taylor Series approximation method of interpolating particle quantities from nodal quantities is based on writing the nodal quantities as a Taylor Series expansion of the quantity at the particle location. Thus, the nodal quantities, such as velocity components in the x-direction, may be written as...

$$u_i = u_p + \frac{\delta u}{\delta x}\Big|_p * dx_i + \frac{\delta u}{\delta y}\Big|_p * dy_i + \frac{\delta^2 u}{\delta x \delta y}\Big|_p * \frac{dx_i{*}dy_i}{2} + D^n\Theta \quad (3.14)$$

where dx_i and dy_i, $i = 1$ to 4, are defined in Eqn. 3.8 to 3.11, $\frac{\delta u}{\delta x}\Big|_p$, $\frac{\delta u}{\delta y}\Big|_p$, and $\frac{\delta^2 u}{\delta x \delta y}\Big|_p$ are the directional derivatives of u at location P and $D^n\Theta$ represent higher order terms.

Equation 3.14, $i = 1$ to 4, is multiplied by weighting coefficients A, B, C, and D, respectively such that the resultant coefficients of u_p are equal to 1 and the resultant coefficients of the directional derivatives are equal to zero. This may be written in matrix form as...

$$\begin{bmatrix} 1 & 1 & 1 & 1 \\ dx_1 & dx_2 & dx_3 & dx_4 \\ dy_1 & dy_2 & dy_3 & dy_4 \\ \frac{dx_1 * dy_1}{2} & \frac{dx_2 * dy_2}{2} & \frac{dx_3 * dy_3}{2} & \frac{dx_4 * dy_4}{2} \end{bmatrix} \begin{bmatrix} A \\ B \\ C \\ D \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \qquad (3.15)$$

This system of linear equations may be solved to yield the weighting coefficients A, B, C, and D. Finally, the particle quantity, u_p, may be written as...

$$u_p = A * u_1 + B * u_2 + C * u_3 + D * u_4 \qquad (3.16)$$

The 4-point Taylor Series approximation method of interpolation yields excellent results for particle path calculations. As illustrated in Figure 3.3(b), the path accuracy using the 4-point Taylor Series interpolation method is significantly improved over the simple bilinear averaging. However, the computational cost for using the 4-point Taylor Series approximation interpolation is large because the coefficient matrix in Eqn. 3.15 must be inverted to solve for A, B, C, and D. A L-U matrix decomposition with maximal column pivoting linear equation solver was developed for solving the system of linear equations and is listed in Appendix 1.

Lagrange Shape Functions

The Lagrange shape function method is an isoparametric technique that is commonly used in finite element analysis. [26-27] For a quadrilateral grid, the linear Lagrange shape functions, N_i, $i = 1$

to 4, are defined such that $N_i = 1$ at node i and $N_i = 0$ at the other nodes. Then, a particle quantity, such as the velocity component in the x-direction, may be written as...

$$u_p = N_1 * u_1 + N_2 * u_2 + N_3 * u_3 + N_4 * u_4 \qquad (3.17)$$

Because the Lagrange shape function method is an isoparametric technique, it is convenient to define a master element, Figure 3.9, in computational space with natural coordinates (ξ,η), such that the shape is squared and ξ and η range from -1 to +1.

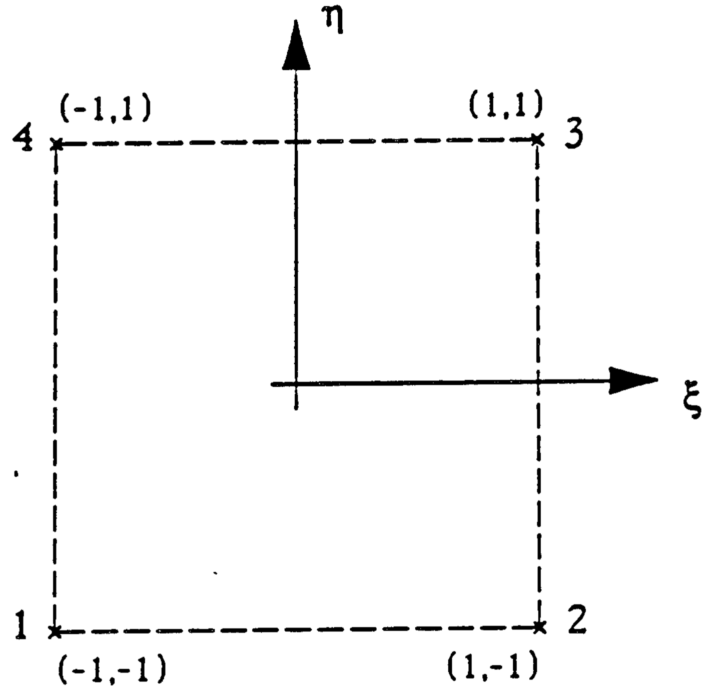


Fig. 3.4 Master element in natural coordinates

The requirement that $N_1 = 1$ at node 1 and $N_1 = 0$ at nodes 2, 3, and 4 is equivalent to requiring that $N_1 = 0$ along the edges $\xi = 1$ and $\eta = 1$. Thus, N_1 may be written as...

$$N_1 = c_1 * (1 - \xi) * (1 - \eta) \tag{3.18}$$

where c_1 is a proportionality constant which is solved for by using the boundary condition that $N_1 = 1$ at node 1 where $\xi = -1$ and $\eta = -1$. Substituting values...

$$N_1 = 1 = c_1 * (1-(-1)) * (1-(-1)) \tag{3.19}$$

yields $c_1 = \frac{1}{4}$. A similar approach may be used to solve for the constants c_2, c_3, and c_4 which yields the desired equations for N_i, $i = 1$ to 4.

$$N_1 = \frac{1}{4} * (1 - \xi) * (1 - \eta) \tag{3.20}$$

$$N_2 = \frac{1}{4} * (1 + \xi) * (1 - \eta) \tag{3.21}$$

$$N_3 = \frac{1}{4} * (1 + \xi) * (1 + \eta) \tag{3.23}$$

$$N_4 = \frac{1}{4} * (1 - \xi) * (1 + \eta) \tag{3.24}$$

Therefore, once the location of the particle has been determined in computational space (ξ,η), the Lagrange shape functions may be solved using Eqn. 3.20 through 3.24 and a particle quantity, such as the velocity, may be computed using Eqn. 3.17.

To determine the location of the particle in computational space requires solving the set of equations...

$$x_p = N_1 * x_1 + N_2 * x_2 + N_3 * x_3 + N_4 * x_4 \tag{3.25}$$

$$y_p = N_1 * y_1 + N_2 * y_2 + N_3 * y_3 + N_4 * y_4 \tag{3.26}$$

for ξ and η. For an orthogonal quadrilateral grid, this reduces to...

$$\xi = \frac{x - \frac{x_1 + x_2}{2}}{\frac{x_2 - x_1}{2}} \tag{3.27}$$

$$\eta = \frac{y - \frac{y_1 + y_2}{2}}{\frac{y_2 - y_1}{2}} \tag{3.28}$$

However, for a non-orthogonal quadrilateral grid, ξ and η cannot be easily equated in terms of the particle and nodal coordinates. In such cases, ξ and η may be determined by using an iterative scheme.

The Lagrange shape function method for interpolating particle velocities from nodal velocities yields excellent results for particle path calculations. As shown in Figure 3.3(c), the path accuracy for the Lagrange shape function method of interpolation is comparable to the accuracy achieved using the 4-point Taylor Series approximation method of interpolation. Again, this represents a significant improvement in accuracy over the simple averaging method of interpolation. However, for an orthogonal quadrilateral grid, the computational time required for the Lagrange shape function method is significantly less than the time required for the 4-point Taylor Series approximation method because the Lagrange shape function method does not require the use of a linear equation solving routine.

Which Interpolation Method Used?

The interpolation methods comparison was performed in the same manner as the time stepping methods comparison. Six initial points were selected as starting locations and particle traces were computed then displayed. The six initial positions were chosen to cover the various regions of the flow field and the particle traces were drawn using line segments to join the time step vertices. Visual inspection revealed accuracy whereas monitoring the number of time steps combined with a knowledge of the particular method revealed computational cost. The results of the interpolation methods comparison is given in Table 3.2

The Lagrange shape function method was chosen for the interpolations of particle quantities from nodal values in this work. This method was chosen for its level of accuracy with respect to computational cost. Both the Lagrange shape function method and the 4-point Taylor Series approximation method significantly improve the level of accuracy of particle path calculations compared to the simple bilinear method of interpolation. However, for a orthogonal quadrilateral grid, the Lagrange shape function method achieves the improvement in accuracy over the simple bilinear averaging without increasing the computational cost while the Taylor series approximation method requires a linear equation solving subroutine.. Thus, for a orthogonal quadrilateral grid, the Lagrange shape function method is clearly the "best" method of interpolation for the particle path calculations.

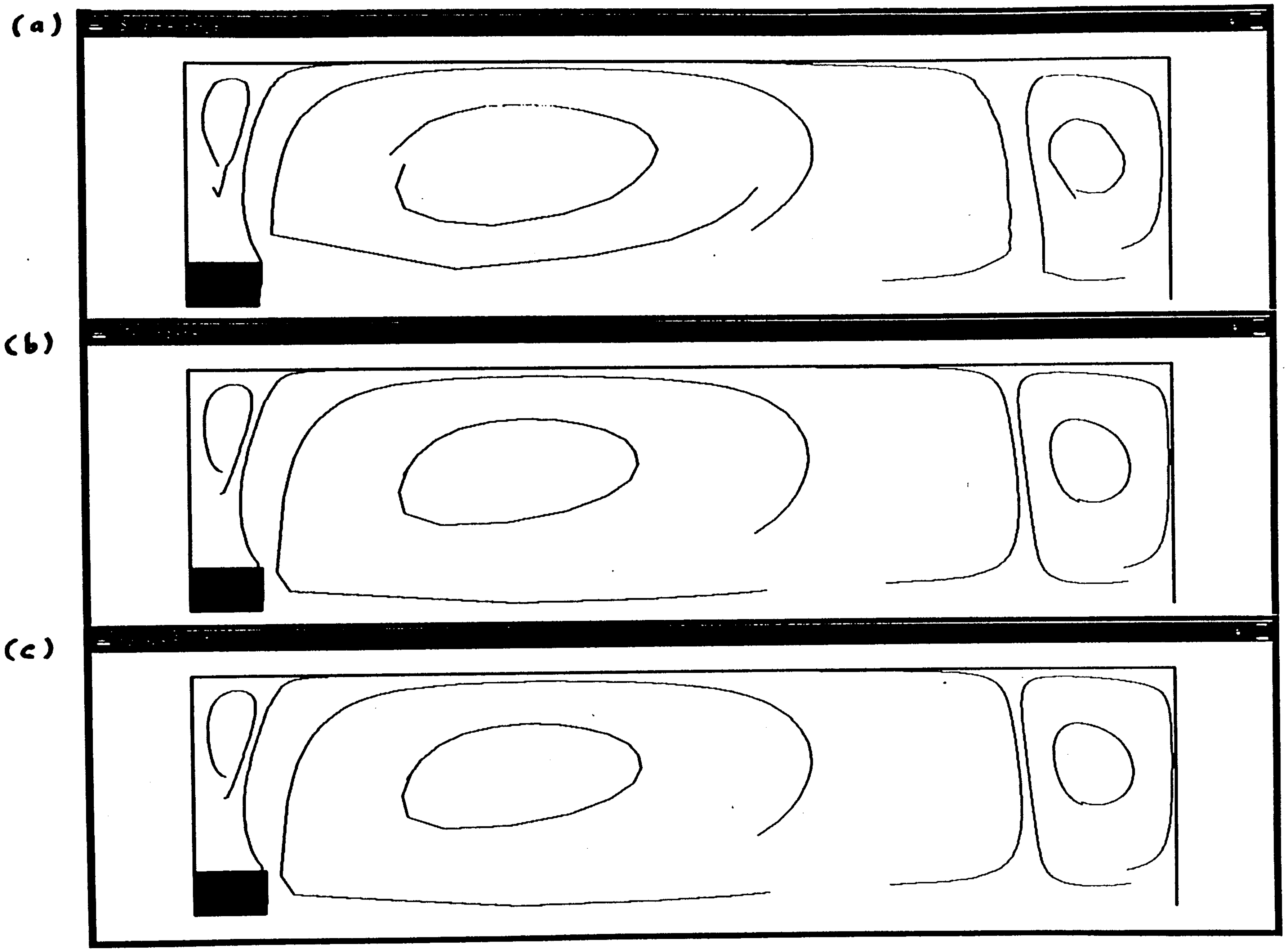


Fig. 3.3 Interpolation methods comparison: (a) Simple Bilinear Averaging (b) 4-pt Taylor Series Approximation (c) Lagrange shape functions

Table 3.2 : Interpolation Methods Comparison

Data File : mint.bin	
i-nodes (x-direction) : 46	(xmin,xmax) = (0.0,0.3556) m
j-nodes (y-direction) : 24	(ymin,ymax) = (0.0,0.0445) m
Linear Shape Function Interpolation	
Optimum time step, dt= 7.347 e-2 s	

starting location (i , j) (x , y) m	# time steps		
	Bilinear Simple Averaging	Taylor Series Approximation	Lagrangian Shape Function
(23,13) (.0936,.0188)	52	46	46
(40,12) (.3388,.0136)	20	19	19
(3,12) (.0118,.0136)	676	810	810
(42,6) (.3485,.0041)	529	488	488
(3,16) (.0118,..0373)	257	232	232
(22,5) (.0789,.0026)	761	829	829

Stopping Criteria

Calculating the path of a Lagrangian particle in an Eulerian flow field involves a series a small particle movements from the initial location using the chosen time stepping and interpolation methods. Any iterative process, such as the particle path calculation, naturally requires a stopping criteria. However, determining when to end the calculation of a particle path is not straight forward.

Depending on the type of flow field used as a data set, the criteria for stopping a path calculation may be quite different. For external flows over a body or internal flows through a pipe or duct, the calculation of a particle path would end when the particle has left the flow field through a defined exit plane. Thus, the stopping criteria simply involves monitoring the particle's location with respect to the exit plane.

However, for steady state, recirculating flow fields, where the calculated pathlines are also the flow streamlines, the particle does not exit the flow field. In such cases, the particle should recirculate and eventually reach the original starting location. Therefore, the stopping criteria would involve monitoring the particle's location with respect to the initial location. Unfortunately, the particle rarely returns to its exact initial location due to numerical inaccuracies in the CFD solution and in the path calculation. Thus, a particle may spiral or overshoot and the computer will fail to recognize the end of the particle path.

Furthermore, a particle may enter a stagnation zone, where the velocity of the flow field is zero or negligible. In such cases, the particle will not move at all or move a negligible distance over a large number of time steps. To further calculate the path of such a particle would not be useful and represents wasted computer time. Therefore, particle stagnation must be recognized so that the path calculation may be terminated.

Thus, the stopping criteria for a particle path calculation must be robust enough to handle a variety of flow situations. The following sections address the stopping criteria developed for recirculating flows and particle stagnation.

Recirculating Flows

In theory, the stopping criteria for a particle path calculation in a recirculating flow field is very simple. End the calculation when the particle returns to its initial location. However, a particle will rarely return to its exact initial location during a path calculation because of the numerical inaccuracies in the CFD solution and the path calculation. Thus, in practice, the stopping criteria for a particle path calculation in a recirculating flow field must compensate for the difference between the initial and final location.

One approach for overcoming the spiraling caused by the numerical inaccuracy is to end the path calculation when the particle returns to within a specified distance of the initial location. Thus, the path calculation would terminate when the distance between the current particle location and the initial particle location becomes less than a specified tolerance, ε. However, a reasonable value of ε is highly dependent upon the path length. Furthermore, even if ε was defined as a function of path length, situations will arise where the particle still may not return to within the tolerance region.

An alternative approach was developed in this work which performs well as a stopping criteria for particle path calculations in a recirculating flow field. After the first particle time step, a line, L_o, is defined perpendicular to the line segment connecting the initial particle location and the current particle location. With reference to Figure 3.11, the particle path calculation may be terminated when the particle has twice intersected the line L_o. To recognize when the particle intersects the line L requires simple linear algebra. The equation of a line may be written as $y = a^*x + b$, where a is the slope and b is the y-intercept. Therefore, if the initial particle location is denoted as (px_1,py_1) and the particle location after one time step is (px_2,py_2), then the line L_o is defined by $y = a_o{}^*x + b_o$, where...

$$a_o = -\frac{px_2 - px_1}{py_2 - py_1} \tag{3.29}$$

$$b_o = (py_1 - px_2)^* a_o \tag{3.30}$$

Similarly, if the current particle location is denoted as (px_i,py_i) and the previous particle location is denoted as (px_{i-1},py_{i-1}), then the line L_i is defined by $y = a_i{}^*x + b_i$, where...

$$a_i = \frac{py_i - py_{i-1}}{px_i - px_{i-1}} \tag{3.29}$$

$$b_i = (py_i - px_i)^* a_i \tag{3.30}$$

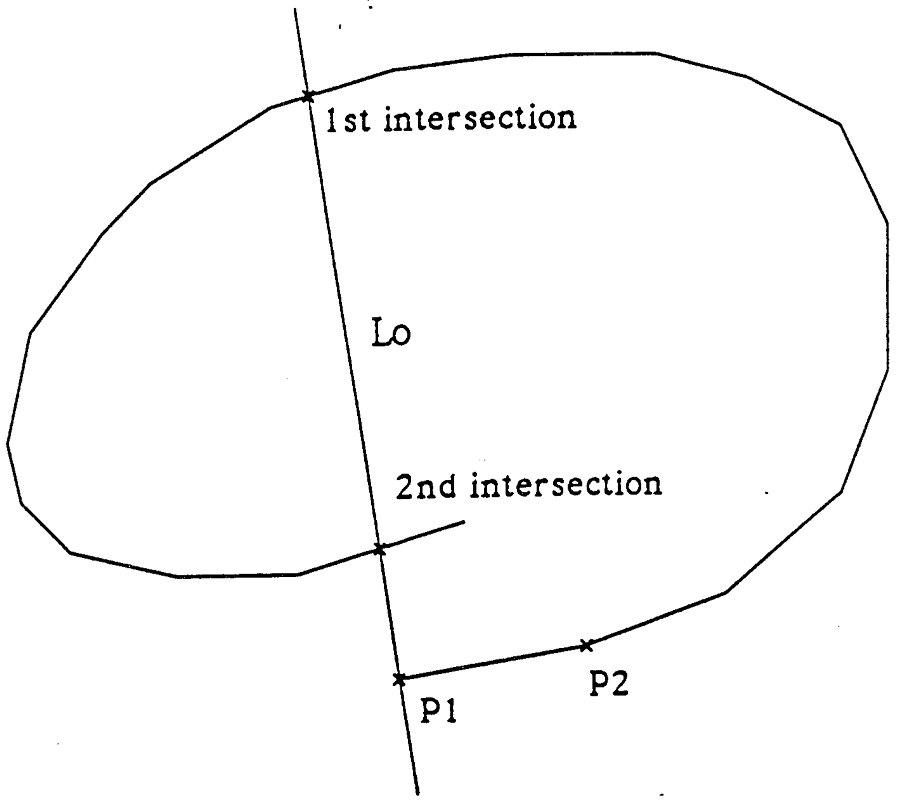


Fig. 3.5 Stopping criteria for recirculating flows

Mathematically, two lines intersect if the determinant of their coefficient matrix is non zero. Therefore, by rewriting L_o and L_i in matrix form as...

$$\begin{bmatrix} 1 & -a_i \\ 1 & -a_o \end{bmatrix} \begin{bmatrix} y \\ x \end{bmatrix} = \begin{bmatrix} b_i \\ b_o \end{bmatrix} \tag{3.31}$$

... L_o and L_i intersect if...

$$\det \begin{vmatrix} 1 & -a_i \\ 1 & -a_o \end{vmatrix} = -a_o + a_i \neq 0 \tag{3.32}$$

Furthermore, the point of intersection is given by Cramer's Rule as...

$$x_i = \frac{b_o - b_i}{-a_o + a_i} \quad (3.33)$$

$$y_i = \frac{-a_o * b_i + a_i * b_i}{-a_o + a_i} \quad (3.34)$$

Thus, if the lines L_o and L_i are found to intersect and the intersection point (x_i, y_i) lies within the range of the line segment Li, then the particle has crossed the line L_o. After the second occurrence of the particle crossing the line segment L_o, the particle path calculation may be terminated. The distance between the initial particle location and the final particle location may be calculated and reported as a measure of the numerical error in the CFD solution and the particle path calculation.

This stopping criteria for a particle path calculation in a recirculating flow field can be extended to 3-D particle path calculations. However, instead of defining a line L_o, a plane P_o is defined perpendicular to the line segment joining the initial particle location and the particle location after one time step. Then, the particle path calculation may be terminated after the particle has twice intersected the plane P_o.

Particle Stagnation

Particle stagnation may occur during a particle path calculation if the particle enters a region of very small, or zero, velocity. When this occurs, the particle will move only a negligible distance over a large number of time steps. To continue the path calculation for a stagnated particle is undesirable because it represents a waste of computer time. A method to recognize particle stagnation must be developed.

A particle is considered stagnated if its velocity components are too small to move the particle a significant distance over a large number of time steps. To recognize particle stagnation requires defining the insignificant distance of travel and the large number of time steps. Because, these values will depend on the particular flow field of interest, they will be defined by using parameters of the flow field.

An insignificant distance of travel is defined as a portion of the total length of the flow field. With respect to the maximum and minimum geometric locations of the flow field...

$$\mathrm{xdist} = (\mathrm{xmax} - \mathrm{xmin}) \tag{3.35}$$

$$\mathrm{ydist} = (\mathrm{ymax} - \mathrm{ymin}) \tag{3.36}$$

$$\mathrm{dstag} = (1.0\ \mathrm{E{-}6}) * (\mathrm{xdist}^2 + \mathrm{ystag}^2)^{1/2} \tag{3.37}$$

Similarly, a large number of time steps is defined as a portion of the maximum number of time steps allowed for a particle path calculation, pmax.

$$\mathrm{mstag} = \mathrm{pmax}/10 \tag{3.38}$$

Thus, for a particular time step, if the distance traveled is less than dstag, then increment the stagnation counter, nstag, by one; when nstag is greater than mstag, end the particle path calculation.

Boundary Crossing

Due to the numerical inaccuracies of the CFD solution and the particle path calculations, a path may be calculated which contradicts physical reality by crossing a surface which has been defined as a solid wall, or boundary, in the flow field. To compensate for this particular problem, the following algorithm has been developed.

1. Determine if the particle has crossed a boundary defined as a solid wall in the flow field.
2. If yes, then move the particle again from the previous location, but only half the distance to the boundary.

In this manner, a particle will never cross a boundary, but may stagnate there, which is physically acceptable and may be appropriate. However, it is important to alert the user that a particle movement has been calculated which crosses a solid wall boundary in the flow field. In doing so, the user is informed of the numerical error which may lie in the CFD solution or the particle path calculation and the animated particle trace will not simply "hide" the error.

Particle Trace Algorithm

The following algorithm was developed to trace the path of a fluid particle in a 2-D, steady state, recirculating flow field. A Huen predictor/corrector time stepping method and a Lagrange shape function interpolation method were chosen for the particle trace algorithm.

```
1.  Choose the initial particle location, $(px_1, py_1)$.
2.  Set path time step counter, $i = 1$.
3.  Do while stopping criteria not met.
    a.  Increment time step counter, $i = i + 1$.
    b.  Interpolate current particle velocity, $(pu_{i-1}, pv_{i-1})$.
    c.  Calculate intermediate particle location.
        i.   $px_{int} = px_{i-1} + pu_{i-1} * dt$
        ii.  $py_{int} = py_{i-1} + pv_{i-1} * dt$
    d.  Check solid wall boundary crossing.
    e.  Interpolate intermediate particle velocity, $(pu_{int}, pv_{int})$.
    f.  Update particle location.
        i.   $px_i = px_{i-1} + \frac{pu_{i-1} + pu_{int}}{2} * dt$
        ii.  $py_i = py_{i-1} + \frac{pv_{i-1} + pv_{int}}{2} * dt$
    g.  Check solid wall boundary crossing.
    h.  Check stopping criteria.
4.  Stop the particle trace.
```

Chapter 4: Mechanics of Computer Graphics

The particle trace animation software developed in this work was written for a Silicon Graphics IRIS-4D GT workstation, which supports the Silicon Graphics Graphics Library (GL). The GL is a UNIX based library of graphics subroutines which facilitates the programming of 2-D and 3-D color graphics and animation. Subroutines are available which provide the user with numerous capabilities related to controlling the graphical environment. The following capabilities were employed to animate Lagrangian particles in a CFD, Eulerian flow field...

1. Drawing Graphical Primitives
2. Higher Order Drawing through the use of NURBS
3. Creating Graphical Objects and Double Buffering
4. Controlling Colors and Color Mode
5. Shading
6. Blending

Drawing Graphical Primitives

Graphical primitives are the basic geometric elements of computer graphics programming and consist of points, lines, and polygons. A series of graphical primitives may be used to create almost any complex shape. A graphical primitive geometry is described by specifying its edges and corners with a list of vertices. A vertex defines the coordinates of a position, or point, in space, and connected vertices form an edge. Edges may be connected as lines to form a wire frame geometry or connected as polygons to form a solid face geometry. The beginning and end of a vertex list is marked by special commands which specify the type of graphical primitive.

The features of graphical primitives may be controlled through different colors and techniques. A point must be assigned a size and a color. A line must be assigned a linestyle, which specifies its width and linetype (solid, dashed, or dotted), as well as a color. In addition, line primitives may be further categorized as polylines, series of connected line segments, and closed lines, polylines with connected first and last points. A polygon must be assigned a fill pattern which controls how and what colors are displayed within the edges of the polygon. Polygons are categorized as simple or non-simple, depending on whether their edges cross, and convex or concave, depending on whether a line segment joining any two points within the polygon is completely contained within the polygon.

Graphical primitives were primarily used in this work to display the flow field geometry. An algorithm was developed to recognize geometry boundaries, or solid walls, and draw them using lines. Similarly, an algorithm was developed to draw the CFD grid using lines that connect the nodes. The line widths and colors defined for the solid walls and the CFD grid were chosen on the basis of visual aesthetics. In addition to the flow field geometry, graphical primitive lines were used to connect the calculated points of a particle trace for comparison of the time stepping and interpolation methods.

Higher Order Drawing through the use of NURBS

Complex curved lines and surfaces may be created through the use of Non-Uniform Rational B-Splines (NURBS). NURBS are based on the principle that a curve or surface may be represented with a series of parametric polynomials joined in a piecewise fashion to create a B-spline. A B-spline is shaped through the use of control points, a basis function, and knots.

In the parametric representation, coordinate of a curve or surface are explicit functions of a parameter. Thus, the coordinates (x,y,z) of curve may be written as a polynomial function of a parameter s...

$$x(s) = a_0 + a_1*s + a_2*s_2 + a_3*s_3$$
$$y(s) = b_0 + b_1*s + b_2*s_2 + b_3*s_3$$
$$z(s) = c_0 + c_1*s + c_2*s_2 + c_3*s_3$$

where the a_i's, b_i's, and c_i's are known as the control coefficients, and the degree of the polynomial is the highest exponent (in this case, 3). Several of these polynomials may be joined in a piecewise fashion to yield higher flexibility in the shape of the curve.

A B-spline is a method to connect and represent the piecewise polynomials describing a curve or surface. A set of control points act as attractors to pull the curve or surface into the desired shape. Control points, which are simply defined by spatial coordinates, form a control polygon which influences the shape of the B-spline, as in Figure 4.1 A basis function determines how the control points influence the curve or surface. For a B-spline curve, the curve segments represented by the parametric polynomials, need not pass through the control points, but must be continuous in their first and second derivatives at the control points where the curve segments join. A basis function is usually represented by a basis matrix which acts on the control points to determine the shape of the curve. Lastly, knots define the relative degree of magnitude of the basis function's influence on particular control points. In this way, knots influence the spline by acting as attractors. Knots may even be defined to force the spline to pass through specific control points.

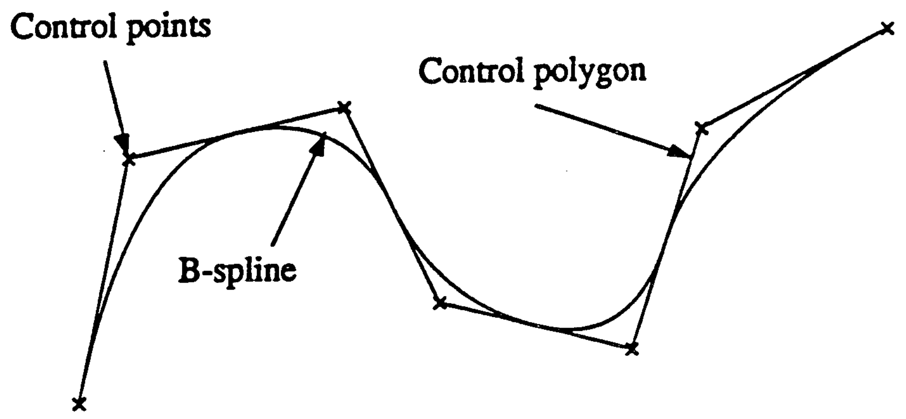


Figure 4.1: Control polygon for a B-spline

NURBS were used in this work to represent the fluid particles for the animation sequences. During a particle trace calculation, the points, or vertices, defining the path of a fluid particle are stored in arrays, px(np) and py(np), where...

px(i) = x-coordinate of particle location at time step i

py(i) = y-coordinate of particle location at time step i

np = total number of time steps calculated for a particle trace.

To represent a fluid particle travelling along the calculated path, a B-spline is drawn using the vertices of the first four time steps, px(i) and py(i), i = 1 to 4, as control points. Then, the B-spline is erased and a new B-spline is drawn using the vertices px(i) and py(i), i = 2 to 5, as control points. This procedure of updating the B-spline which represents the fluid particle by one time step is repeated until i = np.

Using B-splines to represent the fluid particles is advantageous for several reasons. The B-splines smoothe the particle trace between time steps while still ensuring that the fluid particle passes through the calculated vertices for a particle path. Also, each successive B-spline overlaps the previous B-spline by three timesteps and helps to provide continuous motion during animation. Furthermore, the length of the B-spline which represents the fluid particle is directly proportional to the fluid particle's velocity, enabling the user to easily visualize accelerations in the flow field.

Graphical Objects and Double Buffering

Graphical primitives and NURBS are drawing subroutines which act in immediate mode. In other words, the computer immediately displays the images when the subroutines are called. However, it is often more convenient to define a list of graphical subroutines that may be implemented at any time. The use of graphical objects provides this capability and is particularly convenient for computer animation. To define a graphical object, the user marks the beginning and end of the drawing subroutines with special commands that identify the object. Then, the user may implement the drawing subroutines with a command that identifies and calls the graphical object. Furthermore, graphical objects may be edited and changed from anywhere within the graphics program.

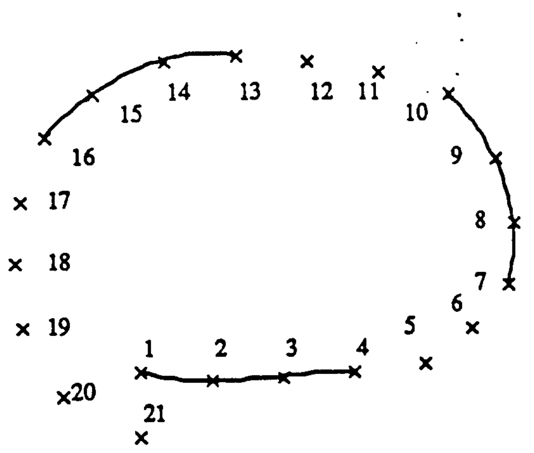
Animation in computer graphics is achieved through the use of double buffering. By default, the display mode in the GL is single buffer mode, where drawing subroutines are written into a single frame buffer which is always visible. However, to simulate the smooth motion required for computer animation, a pair of framebuffers is employed. In double buffer mode, the currently visible frame buffer is labeled the frontbuffer and the drawing frame buffer is labeled the backbuffer. Drawing subroutines are first written into the backbuffer and then the frontbuffer and backbuffer are swithched, so that a completely drawn image replaces the previous one. By drawing into the backbuffer and then "swapping" frame buffers, flickerring due to elapsed time for the execution of the drawing routines is drastically reduced.

Graphical objects and double buffering were used in this work to efficiently animate the fluid particles, represented using NURBS, along their calculated paths. As stated previously, difficulties with computer speed and memory arise when animating particle traces. Because of the lack of CPU speed required for real-time animation of particle traces, the particle trace calculations needed to be separated from the particle trace animation sequences. However, it was still undesirable to store the particle trace information, consisting of spatial coordinates for every time step, in a data base for use as the animation data set. Through the combined use of graphical objects and double buffering, with a thoughtful consideration of traditional frame-by-frame filming techniques, particle trace animations were generated with large savings in computational efficiency.

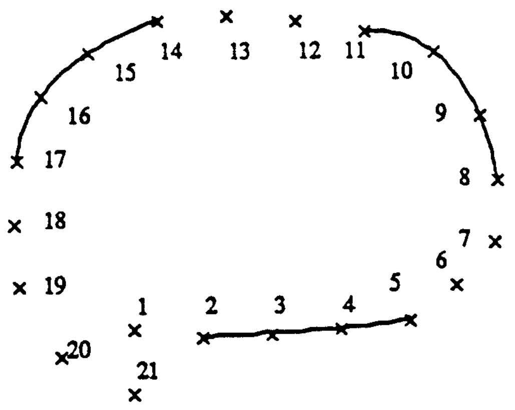
The approach to animating particle traces which was developed in this work separates the particle trace calculation from the animation but does not require storing the particle trace information as a data set by instead storing the particle trace information as a series of images, or graphical objects. However, it would still be computationally unfeasible to store a graphical object for every time step of every particle trace, particularly if numerous particle traces are to be simultaneously animated. Thus, the approach to animating particle traces must use a finite and managable number of graphical objects. This was achieved by animating several fluid particles, each represented as a B-spline through four successive time steps, along the same calculated path separated by a distance of nobj timesteps, where nobj is the defined number of graphical objects.

To illustrate this principle, let the total number of time steps for a particle trace equal thirty-five (np = 21), and let the set number of graphical objects equal seven (nobj =6). Then, the particle trace may be represented as a series of six graphical objects, where fluid particles are drawn using B-splines separated a distance of nobj time steps. Figure 4.2.a through Figure 4.2.f are the six graphical objects which together represent the particle trace. If these graphical objects are shown in rapid sequence, the visual effect is to percieve three ($\frac{21-3}{6}$ = 3) separate fluid particles traversing the entire length of the calculated particle trace. Then, each time a new particle trace is calculated, the same graphical objects may be edited to include the appropriate fluid particles representing the new particle trace. In this way, the total number of graphical objects never increases.

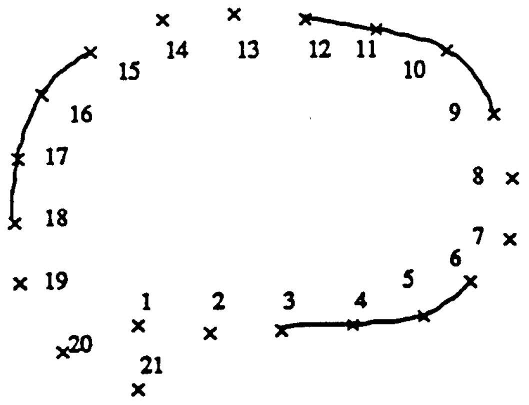
This approach to animating particle traces has several advantageous aspects. By separating the particle trace calculations from the animation sequences and storing the particle traces in a manageable number of graphical objects rather than a data base, large savings are made in computer efficiency in terms of CPU speed and memory. Also, by using a set time step interval of equal magnitude for each particle trace, the relative speeds of the fluid particles is preserved. Furthermore, by animating several fluid particles along the same particle trace, a "fuller" animation sequence to observe the entire flow field is obtained with fewer particle trace calculations.



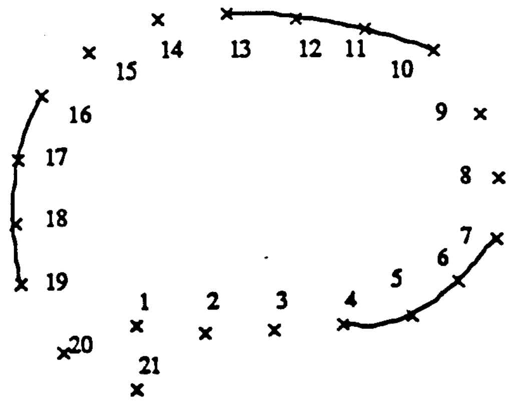
(a) graphical object #1



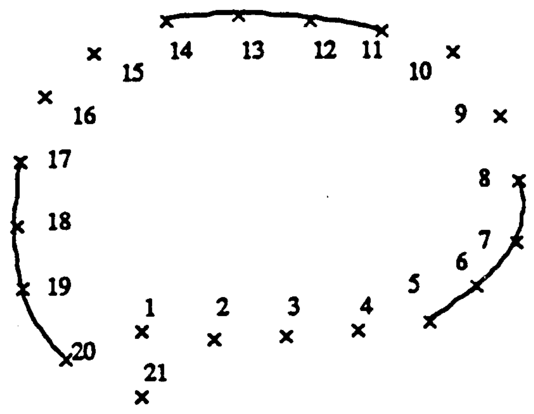
(b) graphical object #2



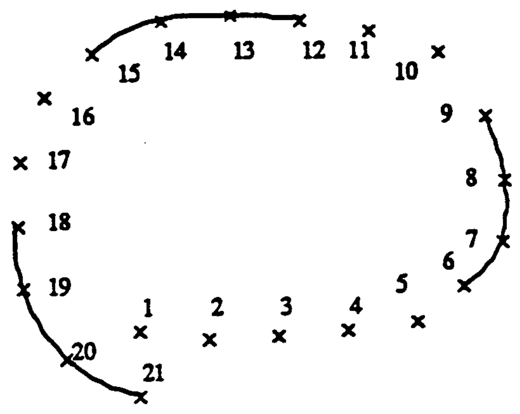
(c) graphical object #3



(d) graphical object #4



(e) graphical object #5



(f) graphical object #6

Figure 4.2: Graphical objects to store particle traces (nobj = 6, np = 21)

However, this approach to animating particle traces has several limitations. If the set number of graphical objects exceeds the number of vertices calculated for a particle trace, then only one fluid particle will traverse the particle trace and it will suddenly disappear when the current graphical object being edited is greater than np. This causes a "pseudo-flickering" in the animation sequence which is observed by a fluid particle suddenly disappearing then reappearing at the beginning of the graphical object cycling. The flickering problem was somewhat eliminated by restarting a new fluid particle at the beginning of the particle trace when $i = \mathrm{int}(\frac{i}{np}) * np + 1$, where i equals the current graphical object being editted. However, slight flickering still occurs because the fluid particles may not completely finish the particle trace within nobj graphical objects due to the ratio $\frac{np-3}{nobj}$ not always equalling an integer value.

Another problem in the animation procedure is related to the variation in calculated path length with respect to the total number of time steps for different particle traces in the same animation sequence. If the path length is relatively small and the total number of computed time steps is relatively large, then the computed time steps will be closely spaced and the fluid particles separated by nobj time steps may actually be indistinguishable. This causes a somewhat cluttered particle trace with excessive color intensity and shimmering where the fluid particles overlap. This problem could possibly be eliminated by allowing nobj to vary for each particle trace and depending upon the path lenth as well as np; however, that would undermine the principle of the animation approach which is based on a preset and unchanged number of graphical objects for the animation sequences. Nevertheless, this problem associated with the approach taken to animating particle traces was determined to be tolerable considering the savings in computational efficiency

Controlling Colors and Color Mode

A standard color monitor for a computer has three color guns that sweep across the entire area of a monitor screen at a specified rate between 60 and 76 times per second. The monitor screen consists of rows and columns of pixels, each containing three separate phosphors that glow red, green, or blue. The intensity of each phosphor at a pixel is controlled by the intensity level of a beam of electrons, between 0 and 255, that the three color guns shoot at the pixel. This is represented in the GL as an RGB triple, where black is (0,0,0) and white is (255,255,255).

Two modes are commonly used for color display -- color map mode and RGB mode. In both color map mode and RGB mode, color information is stored in the bitplanes. A single bitplane consists of exactly one bit of information, a zero or a one, for every pixel on the screen. When creating graphics, the drawing subroutines write information into the bitplanes which the computer hardware then interprets as colors to be displayed at the pixels.

In color map mode, the zeros and ones in the bitplanes are interpreted as a binary number which references an index of a color map look-up table. Color map mode is the default mode in the GL and a default color map is conveniently available. To specify a color, the user simply specifies an integer value of a color map location. The computer then looks up the RGB triple corresponding to that color map location and the RGB triple becomes the current color for all subsequent drawing routines. The user may also change or modify the default color map by calling a subroutine and providing an color map index number as well as the desired RGB triple.

In RGB mode, 8 bits (256 values) are provided for each color component -- red, green, and blue. The user simply specifies a value from 0 to 255 for each color component. This set of values then becomes the RGB triple for the current color and all subsequent drawing routines will display that color on the screen. However, to use RGB mode in the GL, the user must first set the color mode to RGB mode with the appropriate GL subroutine.

The selection of a color mode, color map or RGB, in the particle trace animation software depended upon the desired attributes of the animation sequence. If a color contour of a flow field variable was to be shown as a background image for the animated fluid particles, then color mode was chosen so that a color map could be produced for the contour color levels. However, if different colored fluid particles were animated to represent the various components of a multi-phase flow field, and the color intensity of a fluid particle depended upon volume fraction data for the various components, then RGB mode was chosen so that alpha comparisons could be performed to control color intensity.

In addition, color writemasks which control the writing of color information into the bitplanes were employed to enable any background images (color contours, reactor geometry, or CFD grid) to be drawn only once, prior to the beginning of the animation sequence. To achieve this, the static images were initially drawn into both the frontbuffer and backbuffer, then special writemask commands were issued which write-protected the images from being erased but allowed the fluid particles to visibly pass over the background images.

Shading

Both RGB mode and color map mode provide the capability to define the current color based on respective red, green, and blue intensities. Then, all subsequent drawing routines will be based on the current color. This is sufficient for drawing flat shaded objects of uniform color on the screen. However, this is insufficient for for approximating reality where colors may vary over a surface and shading effects occur. Fortunately, computer graphics hardware calculates shading effects and allows the user to vary color over a surface without specifying the color on a pixel by pixel basis.

A common technique to vary the color across a polygon is through Gouraud shading. By specifying the true color of a polygon at each vertex, the computer will shade the polygon based on those values. Gouraud shading first linearly interpolates the RGB components from vertex to vertex along each edge of the polygon. Then, Gouraud shading linear interpolates the colors from edge to edge across the face of the polygon. Thus, Gouraud shading eliminates the optional method varying the color of an object by using increasing number of flat shaded polygons.

Gouraud shading was used in the particle trace animation software to display backgound images of color contours representing variation in flow quantities. To generate a color contour, a color ramp is first created in color map mode. A corresponding range is also defined which relates the high and low values of the variable under consideration to the far ends of the color ramp. Then, Gouraud shaded polygons are drawn using the vertices of the CFD grid with the color at each vertex defined in terms of the nodal value of the variable under consideration.

In this work, background color contours were used to display simulated attenuation from x-ray radiographs of a closed bath combustion of liquid metal fuels. A numerical method based on x-ray cross-sections and a multifluid CFD code was used to determine the attenuation levels of the individual phase components at a specific percent utilization. The attenuation calculations were then used to generate background color contours for the animated particle traces. Color contours were also used in this work to display the variation of volume fractions of the particular phases in a multiphase flow field.

Blending

Another higher level capability to enhance the performance of color display involves the use of blending. By default, the pixel colors of incoming objects simply replace the current pixel colors where the object is to be drawn. However, to simulate reality it is often desirable to replace the current color at the pixels with a color that is a function of the incoming pixel color and the current pixel color. Blending provides this capability by assigning a blending factor alpha, between 0 and 255, to the bitplanes in addition to the RGB components. Then a blending function may be written in terms of the incoming alpha value to perform an alpha value comparison which determines how to blend the RGB color components. Graphics libraries usually provide a set of blending functions which approximate various physical phenomena such as transparency and material composition.

Blending was used in this work to animate particle traces for multi-phase flows. A CFD data file for a multi-phase flow field contains volume fraction information in addition to the velocity components of the respective phases at the nodal locations. Using the respective velocity components, separate particle traces are calculated for the different phases of a multiphase flow field and the particle traces are distiguished by using different colors for the fluid particles of each phase. Then, a transparency blending function is based on the volume fraction data to scale the color intensity of a fluid particle with respect to the relative amount of a phase present at a particular location. Thus, a fluid particle for a phase has full intensity if 100% of the phase is present and appears completely transparent if no fraction of the phase is at the particle location. By doing so, a flow field is developed with animated particles that reflect the flow fields multiphase composition

Chapter 5: Applied Examples

Particle trace animations were generated for two case studies of CFD flow fields. The "mint" flow field is a 2-D, steady state, single-phase, recirculating flow field which models a closed bath combustion process of a liquid metal fuel (Li). The mint CFD data file was the sample data file used for developing the particle trace calculation and animation software. The mint CFD data file contains the nodal Cartesian coordinates of the CFD grid and the Cartesian velocity components at the respective nodes. The "x-ray" flow field is a 2-D, steady state, multi-phase, recirculating flow field which models a closed bath combustion of a liquid metal fuel. [19] The output data file for the x-ray flow field contains the nodal Cartesian coordinates of the grid, the Cartesian velocity components of three separate phases, the volume fraction data of the phases, the density of the phases, and the x-ray radiograph attenuation levels of the phase mixture.

The particle trace animation software was written in FORTRAN and a listing of the code for the first animated particle trace (Apt1) using the mint flow field is given in Appendix B. The subroutines which were modified for subsequent particle trace animations, including the multiphase flow field, "x-ray", are given in Appendix C. In addition, a video tape of all animated particle traces discussed in this chapter is available through the library of the Applied Research Laboratory in the Applied Science Building at University Park campus, Penn State University.

Mint Flow Field

The mint flow field is a 2-D, steady state, single-phase, recirculating model of a closed bath combustion of a liquid metal fuel. The combustion vessel is symmetric with the injection of fuel and oxidant along the centerline. The principle of symmetry was used in the geometric modelling and the mint flow encompassed only $\frac{1}{2}$ of the combustor. In doing so, the injection of fuel and oxidant occurs in the lower left corner of the flow field geometry. The placement of the injector causes a jet of fluid across the bottom of the flow field and particle trace animations reveal the entrainment of fluid into the jet. This jet then results in the formation of a primary recirculation zone of high velocity in the middle of the flow field and two secondary recirculation zones of low velocity at the outer left and right edges of the flow field.

The mint flow field was modelled computationally with a 46 x 24 quadrilateral, orthogonal grid which is .3556 m wide and 0.04445 m high. For the particle trace animation of the mint flow field, the height of the reactor was scaled 2.5:1 with respect to the width. This caused the mint reactor geometry to fill more space on the computer screen and increased the flow field's perceived sensitivity in the vertical direction. The time step interval for the particle trace calculation mint flow field was established as dt = 7.347E-2 m/s through trial and error to find the maximum dt such that the desired level of accuracy was retained.

The mint flow field was primarily used for the development of the particle trace calculation algorithm and animation software. The time stepping and interpolation methods comparisons for the path calculations (Chapter 3) were done using the mint flow field. Then, the method to animate the particle traces (Chapter 4) was developed for the mint flow field. The mint flow field was then used to generate particle trace animations with differing means of inputting the initial conditions of the particles.

Animated particle trace #1 (Apt1) supplies user input of initial particle location through the mouse cursor and left mouse button. To begin a particle trace, the user positions the mouse cursor in the desired starting location and presses the left mouse button. This initiates a particle trace and then adds the path information to the set number of graphical objects. For Apt1, the number of graphical objects, nobj, was set at 75 to compromise between the wide variation in total number of computed time steps ($np < 20$ in regions of high velocity within the inner recirculation zone and $np > 1000$ in areas of low velocity within the outer recirculation zones.

Apt2 queries input of initial particle location through a user subroutine. The user supplies the number of initial particle locations and the respective x and y coordinates (or i and j node identifiers). The software first performs the particle trace calculations, storing each trace in the graphical objects, then continuously cycles through the graphical objects to animate the fluid particles. By initiating the particle traces from within the source code, numerous traces may be computed with relative ease and a "fully" developed flow field picture may be obtained. Apt2 starts particle traces at numerous locations distributed evenly throughout the flow field. This provides insight into the global flow pattern. Due to the increasing complexity of the graphical objects as the number of particle traces increase, the maximum number of time steps was limited to 300 and the number of graphical objects was increased to 150.

Apt3 also starts the particle traces from within the source code. Apt3 generates a dynamic timeline by starting a vertical column of fluid particles in the center of the flow field. For the timeline, it was desirable to have only a single particle traverse the particle path. This was achieved by eliminating the recirculating flow and stagnation stopping criteria, setting the number of time steps for each particle trace to 100 and also setting the number of graphical objects to 100. By doing so, each graphical object contained one fluid particle for each separate particle trace.

For all animated particle traces with the mint flow field, color map mode was used with writemasks so that the flow field background (reactor walls and CFD grid) only required one draw into both the frontbuffer and backbuffer. This saves considerable computational time during the animation sequence, when computer speed directly affects animation smoothness. The use of writemasks in color map mode disables the background images from being erased while allowing the fluid particles, represented by the color red (255,0,0), to visibly pass over the background image. Then, when the fluid particles are erased for the next graphical object to be drawn, the background images reappear.

X-ray Flow Field

The x-ray flow field is a 2-D, steady state, multi-phase, recirculating CFD model of a closed bath combustion of a liquid metal fuel (Li) at 60% utilization. The primary reason for the modelling of the x-ray flow field is to produce a numerical visualization that can be compared to an actual radiographic video sequence or picture. The details of the actual reactor which the x-ray flow field models is given in the JANNAF paper presented by Dr. Tim Miller of the Applied Research Laboratory, Penn State University. [19] The size of the reactor model is 1.095 m long by.337 m high. A pair of staggered injection ports are placed on the left and right walls of the reactor and impart a swirling momentum to the reactor bath. The combustor is also tilted 45^{o} with respect to the horizontal.

The x-ray flow field was originally modelled as a full 3-D flow field. However, for the particle trace animations, a 2-D model was extracted from the 3-D solution by averaging all flow quantities across the z cross-sections. This is valid because an actual radiograph film or image is a spatial (z) averaged representation of a multiphase flow field. The resulting nodal network was a 29 x 21 quadrilateral, orthogonal grid. The 2-D data file contains the nodal values of velocity for three phases, fuel, product, and vapor, the volume fractions of each phase, and the density of each phase. In addition, x-ray radiographic attenuation data was generated using Eqn. 2.3.

Figure 5.1 is a digitized image of an experimental x-ray radiograph of a closed bath combustion at 60% utilization. Note that the reactor is tilted 45^{o} with respect to the horizontal. Thus, gravity helps to pull the heavier product phase to the lower, right corner of the reactor while allowing the lighter vapor phase to stratify at the upper, left corner. This is shown on the x-ray radiograph by high attenuation values (0.9) in the lower right corner and low attenuation values (0.25) in the upper, right corner, because the product is the most attenuating phase. As seen in the jagged and somewhat chaotic contour lines of attenuation, the instantaneous x-ray radiograph highlights the turbulent nature of the flow field.

Figure 5.2 is a contour plot of the numerically calculated x-ray attenuation levels for the CFD flow field which models the closed bath combustion at 60% utilization. [19] Again, the high attenuation values occur in the lower, left corner of the reactor, indicating high concentration of product phase, and the low values are located near the top, indicating low concentration of product and higher concentrations of vapor. However, the numerical x-ray radiograph does not convey the highly turbulent image of the experimental x-ray radiograph. The discrepancies between the two images is because the experimental x-ray radiograph is an instantaneous image at 60% utilization while the CFD code is based on a time average of the closed bath combustion at 60% utilization.

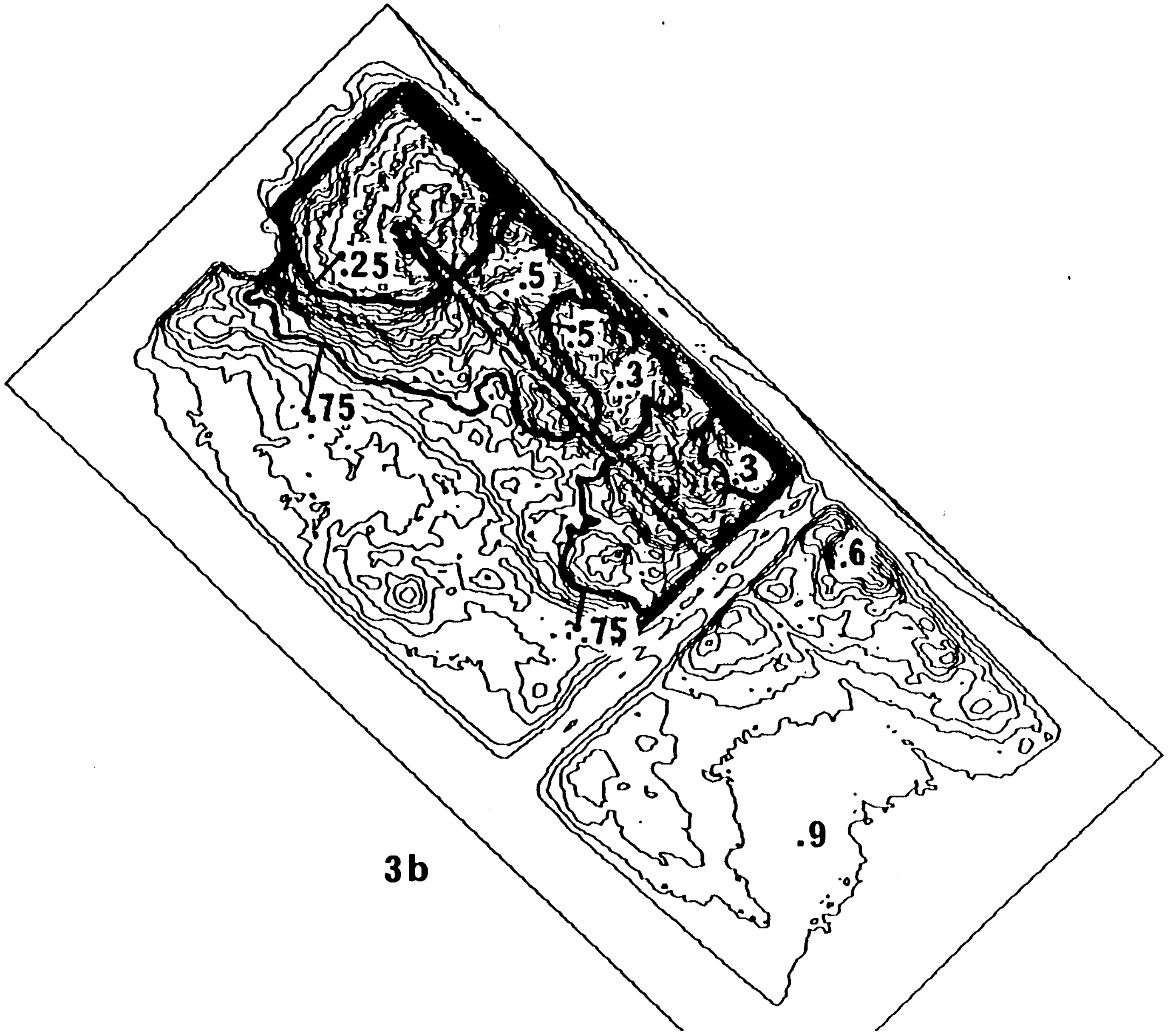


Fig. 5.1 Digitized x-ray radiograph of a closed bath combustor at 60% utilization

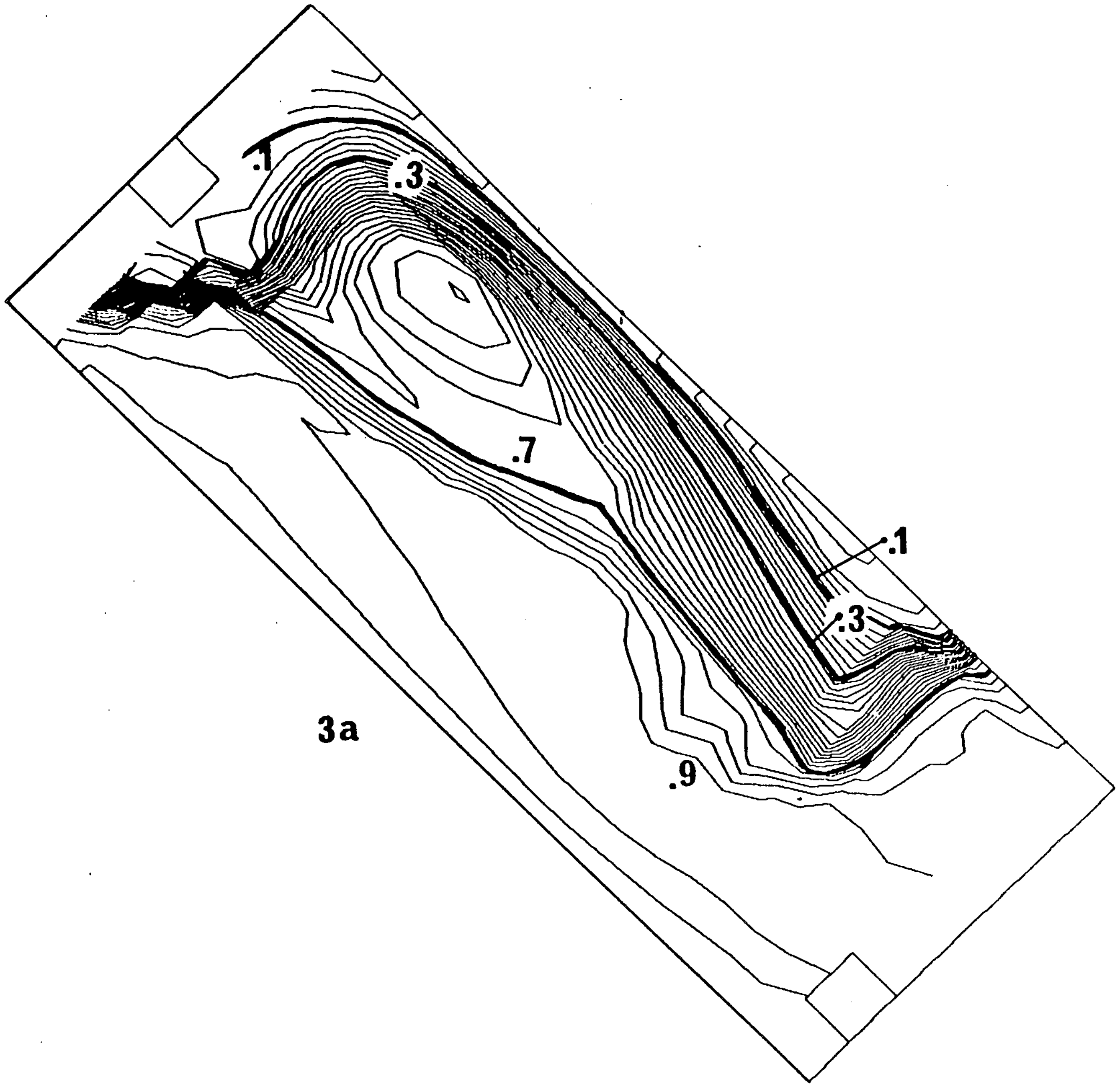
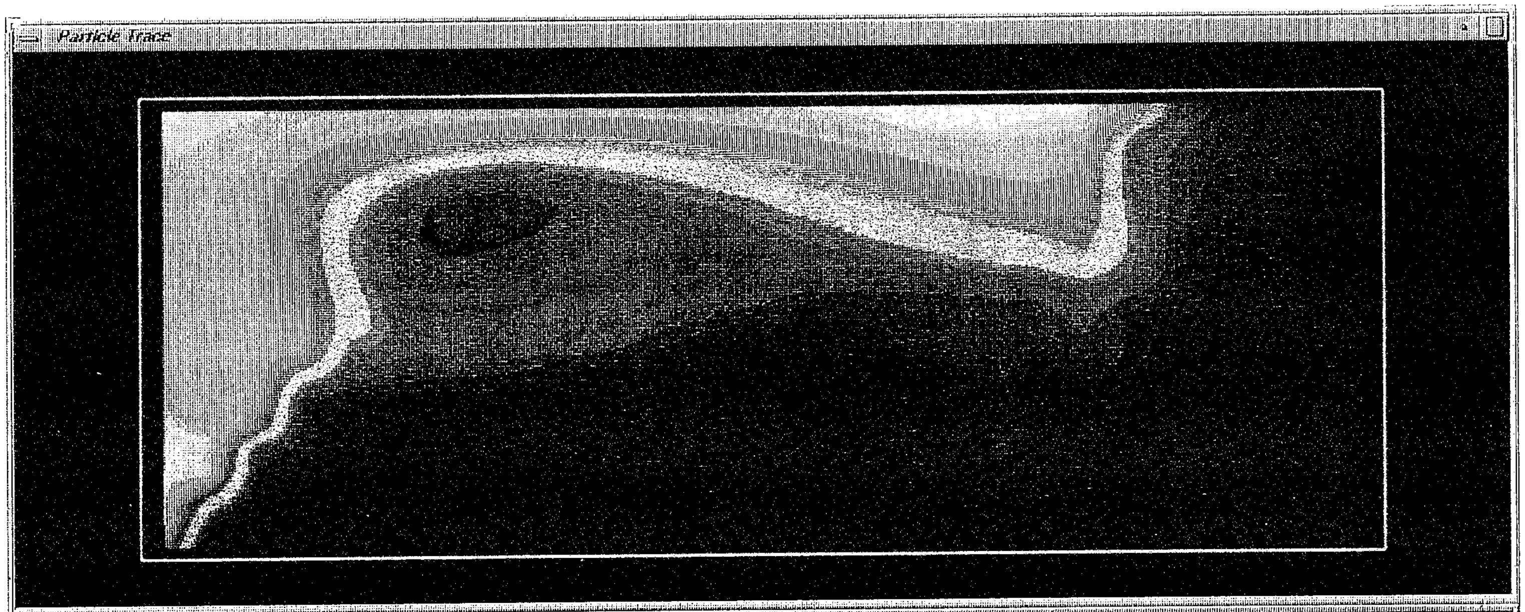


Fig. 5.2 Contours of numerically calculated x-ray attenuation levels for the x-ray flow field

Figure 5.3(a) is a color contour based on the same numerical data set used for Figure 5.2. In Figure 5.5(a), blue (0,0,255) is scaled to the highest possible attenuation level (1.0), green (0,255,0) is scaled to the medium attenuation level (0.5), and red (255,0,0) is scaled to no attenuation (0.0). Attenuation values between 0 and 0.5 interpolate color between red and green while attenuation values between 0.5 and 1.0 interpolate color between green and blue. Figure 5.3(a) compares well to Figure 5.4 and helps to further convey the variation in attenuation levels through the use of false coloring. However, when displaying contours using false coloring, knowledge of the relationship between the color scale and the represented data is essential.

Color contours were also used in this work to display the variation of volume fraction data of a particular phase in a multi-component flow field. Here, blue (0,0,255) is scaled to the maximum volume fraction (1.0), green (0,255,0) is scaled to the mid volume fraction (0.5), and red (255,0,0) is scaled to the minimum volume fraction (0.0). For the 2-D x-ray flow field, Figure 5.3(b) displays the variation in the volume fraction of the fuel species , Figure 5.3(c) displays the variation in the volume fraction of the product species, and Figure 5.3(d) displays the variation in the volume fraction of the vapor species. As seen in these Figures, the product (high density) is concentrated near the bottom of the reactor, the vapor (low density) is concentrated near the top of the reactor, and the fuel (mid density) is primarily mixing near the center of the reactor.

(a)



(b)

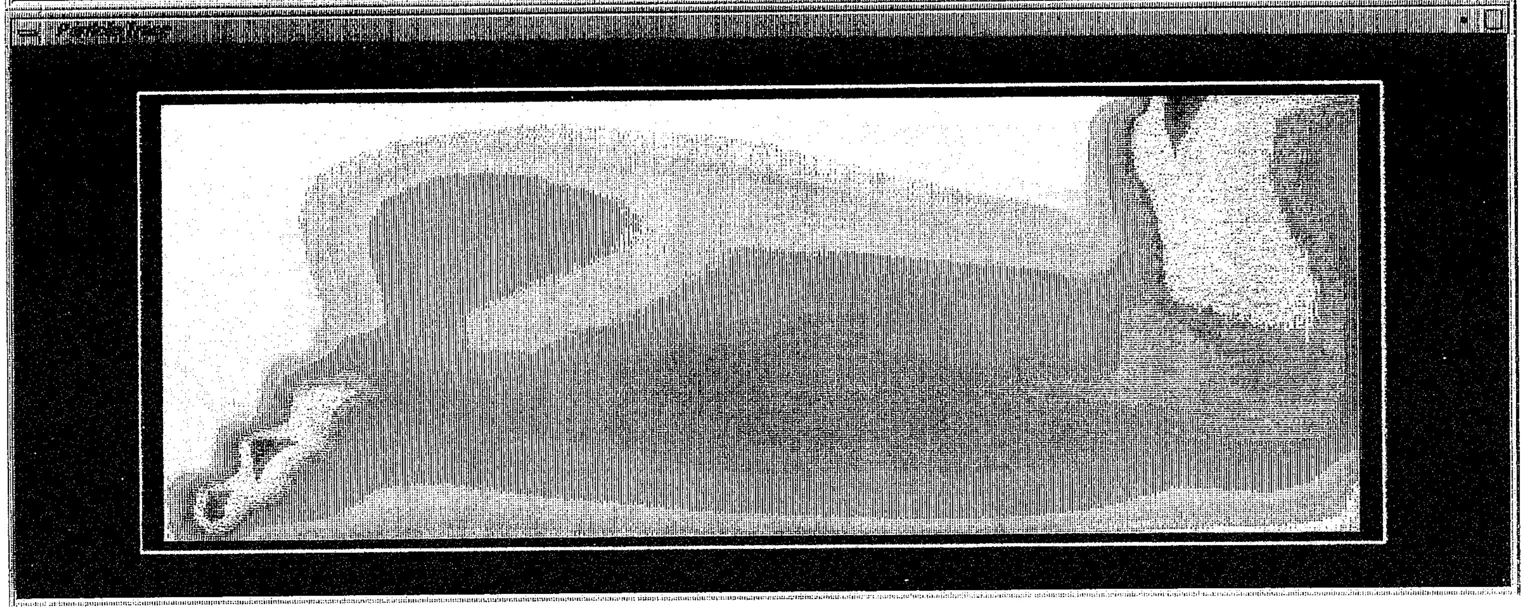
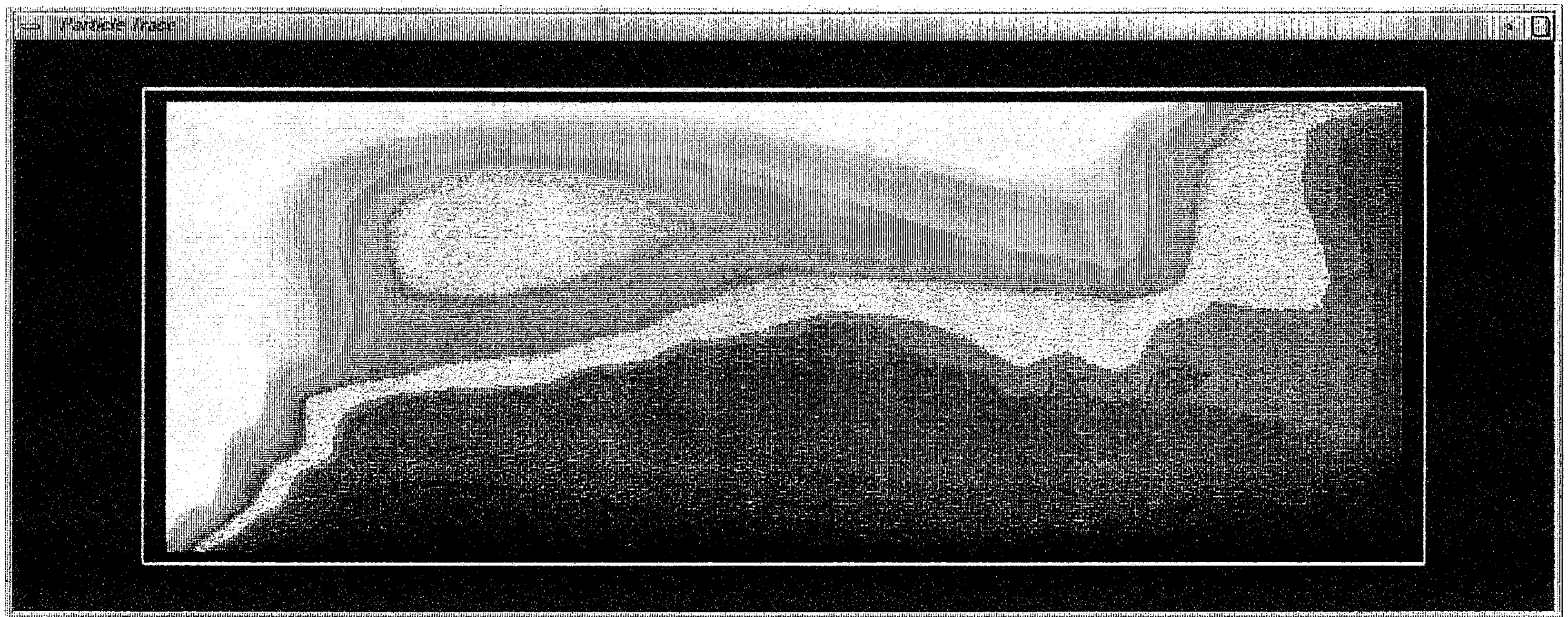
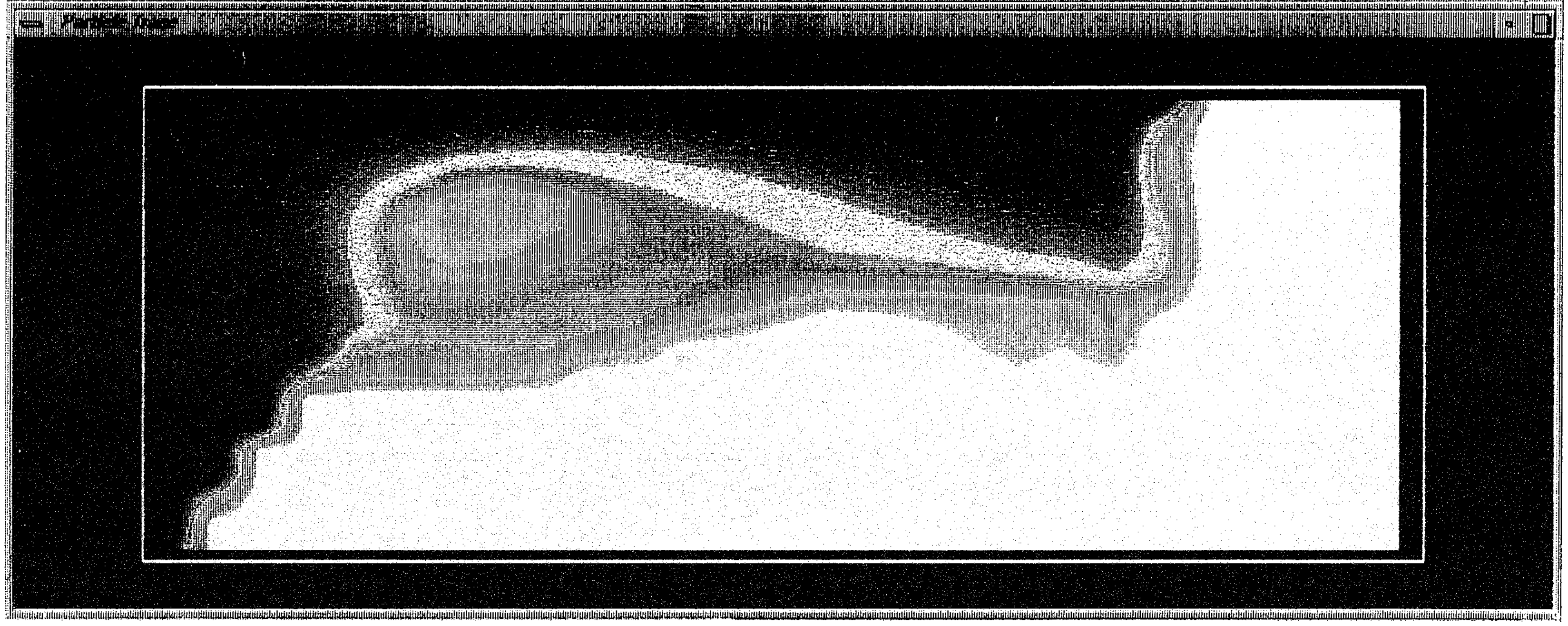


Fig. 5.3 Color contour background images for x-ray particle trace animations: (a) x-ray attenuation levels (b) fuel volume fraction (c) product volume fraction (d) vapor volume fraction

(c)



(d)



To convey the fluid motion of the x-ray flow field, numerous particle traces were developed that highlight various aspects of the multiphase nature of the x-ray flow field. Apt4 displays a gray scale contour of the x-ray attenuation levels as a background and uses the mixture velocity, a volume fraction average of the component velocities, to calculate the particle traces. Apt4 uses solid red particles and provides interactive user input of initial particle location through the mouse. The number of graphical objects was set at 75 while the maximum allowable number of time steps was set at 1000.

Apt5 displays a color contour of the x-ray attenuation levels (Fig. 5.3(a)) as a background and also uses the mixture velocity to calculate the particle traces. White (255,255,255) was chosen for the particles because the background contour used a wide spectrum of colors. A large number of particle traces are calculated for Apt5 by initiating the particle traces from within the source code. Therefore, the number of graphical objects was increased to 100 while the maximum allowable number of time steps was reduced to 200 so that the complexity of the graphical objects did not hinder smooth animation. Apt6 is identical to Apt5, except that the mixture density is used to generate the background color contour. The striking similarity between the two color contours highlights the relationship between the density of a media and its x-ray attenuation levels.

The next series of animated particle traces convey information of the particular phases in the multiphase flow field. Color contours of variation in volume fraction data were used as the background images (Figures 5.3(b)-(d)) and the mass velocity of respective phases were used to calculate the particle traces. Apt7 displays the color contour of variation in volume fraction for the fuel as a background image and uses the fuel mass velocity to calculate the particle traces. The fluid particles are represented as solid white and the user interactively supplies the starting locations with the mouse. Apt8 is identical to Apt7 except that the product phase is visualized and Apt9 is similar except that the vapor phase is visualized. By visualizing particle traces and color contours of volume fraction for individual phases in a multiphase flow field, considerable insight is gained into the relative amount of the phases present and the relative effect of the phases in influencing the overall mixture velocity.

The final animated particle trace, Apt10, of the x-ray flow field computes separate particle traces for each phase present in the multiphase flow field, distinguishes the fluid particles for each phase with respective colors, and controls the intensity of the particle colors with volume fraction data of the particular phases. For Apt10, the fuel is represented as green fluid particles, the product is represented as blue fluid particles, and the vapor is represented as red fluid particles. To control the intensity of the particle colors requires the use of alpha value comparisons which are only available in RGB mode. The background image for the animation sequence of Apt10 only consists of the reactor geometry and the CFD grid. The starting information is supplied within the source code so that numerous particle traces may be visualized. Using multiple phases represented by different colors, provides excellent insight into the multiphase nature of the x-ray flow field. Where little or no amount of a phase exists, the respective fluid particles are nearly invisible and where a large relative percentage of a phase exists, the respective fluid particles are highly visible.

Chapter 6 - Conclusion

In this work, a CFD post processing technique was designed to generate particle trace animations for 2-D, steady state, multiphase, closed, recirculating flow fields. The first step was to develop a suitable particle trace algorithm. Various time stepping and interploation methods were investigated and a Huen predictor/corrector time stepping method combined with a Lagrange shape function interpolation method was chosen for the particle path calculations based on desired level of accuracy attained with low computational cost. Also, a special stopping criteria for recirculating flow fields was developed to help reduce the path spiralling and overshoot which can occur when a fluid particle does not return to its initial location. Computer graphics techniques, implemented on a Silicon Graphics IRIS 4D GT workstation, were next developed to represent and animate the fluid particles along their calculated paths. The fluid particles themselves were represented using B-splines and the combined use of graphical objects with double buffering enabled a computationally efficient particle trace animation program to be developed which separates the path calculations form the path animations and stores the calculated paths in a pre-defined number of images rather than in a data base. Capabilities were also developed to plot scalar contours of flow field quantities as background images for the particle trace animations and to scale the color of the fluid particles according to flow field variables.

These techniques were then applied to a 2-D, steady state, multiphase, closed, recirculating flow field which models a closed bath combustion of a liquid metal fuel. Scalar contours of x-ray attenuation levels were plotted for a multiphase flow field to simulate x-ray radiographs of closed combustion of liquid metal fuels. Volume fraction data of the phases was also used to plot background color contours and to control the individual color intensities of multiple color fluid particles representing the various phases.

Appendix A

Linear Equation Solver

Appendix A : Linear equation solver

L-U decomposition with maximal column pivottin

```
      subroutine le(a,n,x)
*
* input coefficient matrix a(n,n+1) which includes  coefficient matrix b
*
      double precision a(4,5),ml(4,4),x(4),z(4)
      double precision maxp1,maxpi,HOLD,acopy
*
* find first pivot element
*
      np1=n+1
      nm1=n-1
      maxp1=0.0
      do 10 ii=1,n
      if (abs(a(ii,1)).gt.abs(maxp1)) then
      maxp1=a(ii,1)
      ip1=ii
      endif
   10 continue
      if (abs(a(ip1,1)).eq.0.0) then
      print *,'no uniquie soln exists'
      stop
      endif
*
* interchange pivot row ip1 and 1 in a
*
      if (ip1.ne.1) then
      do 25 JJ=1,np1
      acopy=a(1,JJ)
      a(1,JJ)=a(ip1,JJ)
      a(ip1,JJ)=acopy
   25 continue
      endif
*
* define first row of U and first column of L in a
*
      ml(1,1)=1.0
      a(1,1)=a(1,1)/ml(1,1)
      do 30 JJ=2,n
      a(1,JJ)=a(1,JJ)/ml(1,1)
      a(JJ,1)=a(JJ,1)/a(1,1)
      print *,'a(1,',JJ,') =',a(1,JJ)
      print *,'a(',JJ,',1) =',a(JJ,1)
   30 continue
      do 40 ii=2,nm1
      iip1=ii+1
      iim1=ii-1
*
* find ith pivot
*
      maxpi=0.0
      do 50 JJ=ii,n
      sum1=0.0
      do 60 kk=1,iim1
      sum1=sum1+a(JJ,kk)*a(kk,JJ)
   60 continue
```

```
      print *,'sum1 = ',sum1
      if((abs(a(jj,ii)-sum1)).gt.maxpi) then
      maxpi=abs(a(ii,jj)-sum1)
      ipi=jj
      endif
   50 continue
      print *,'maxpi = ',maxpi
      print *,'ipi = ',ipi
      if (maxpi.eq.0.0) then
      print *,'no unique soln exists'
      stop
      endif
      print *,'ii =',ii
      if (ipi.ne.ii) then
*
* interchange rows ipi and ii in both A and L
*
      do 70 jj=1,np1
      acopy=a(ii,jj)
      a(ii,jj)=a(ipi,jj)
      a(ipi,jj)=acopy

      print *,'a(',ii,',',jj,') = ',a(ii,jj)
      print *,'a(',ipi,',',jj,') = ',a(ipi,jj)
   70 continue
      endif
*
* determine ith row of U and ith column of L
      ml(ii,ii)=1.0
      sum2=0.0
      do 80 kk=1,iim1
      sum2=sum2+a(ii,kk)*a(kk,ii)
   80 continue
      a(ii,ii)=(a(ii,ii)-sum2)/ml(ii,ii)
      print *,'sum2 = ',sum2

      print *,'a(',ii,',',ii,') = ',a(ii,ii)
      do 90 jj=iip1,n
      sum3=0.0
      sum4=0.0
      do 100 kk=1,iim1
      sum3=sum3+a(ii,kk)*a(kk,jj)
      sum4=sum4+a(jj,kk)*a(kk,ii)
  100 continue
      a(ii,jj)=(a(ii,jj)-sum3)/ml(ii,ii)
      a(jj,ii)=(a(jj,ii)-sum4)/a(ii,ii)
      print *,'sum3 =',sum3
      print *,'a(',ii,',',jj,') = ',a(ii,jj)
      print *,'sum4 =',sum4


      print *,'a(',jj,',',ii,') = ',a(jj,ii)

   90 continue
   40 continue
```

```
      sum5=0.0
      do 110 kk=1,nm1
      sum5=sum5+a(n,kk)*a(kk,n)
 110  continue
      HOLD=a(n,n)-sum5
      print *,'sum5 =',sum5
      print *,'hold =',HOLD
      if (HOLD.eq.0.0) then
      print *,'no unique solution exists'
      stop
      endif
c
c solve lower tri system Lz=b
*
      ml(n,n)=1.0
      a(n,n)=HOLD/ml(n,n)
      z(1)=a(1,np1)/ml(1,1)
      print *,'a(4,4) = ',a(n,n)
      print *,'z(1) = ',z(1)
      do 120 ii=2,n
      iim1=ii-1
      sum6=0.0
      do 130 jj=1,iim1
      sum6=sum6+a(ii,jj)*z(jj)
 130  continue
      z(ii)=(a(ii,np1)-sum6)/ml(ii,ii)
      print *,'sum6 = ',sum6
      print *,'z9',ii,') = ',z(ii)
 120  continue
c
c solve upper tri system Ux=z
      x(n)=z(n)/a(n,n)
      do 140 ii=nm1,1,-1
      iip1=ii+1
      sum7=0.0
      do 150 jj=iip1,n
      sum7=sum7+a(ii,jj)*x(jj)
 150  continue
      print *,'sum7 =',sum7
      x(ii)=(z(ii)-sum7)/a(ii,ii)
 140  continue
      do 160 ii=1,n
      print *,'x(',ii,') = ',x(ii)
 160  continue
      end
```

Appendix B

Source Code for Apt1

```
      program apt1
************************************************************************
*                                                                      *
* Ted Blackmon, GFW                                          4/15/92  *
* Applied Researh Lab                                                  *
* Penn State University                                                *
*                                                                      *
* Animated particle traces for a 2-D, steady state, single phase,     *
* recirculating flow in a closed combustor of a liquid metal fuel     *
*                                                                      *
*                                                                      *
*   -- Starting location of particle input interactively through left *
*      mouse button                                                    *
*   -- Linear shape function velocity approximation                    *
*   -- Huen predictor/corrector time stepping method                   *
*   -- Background is the coordinate grid                               *
*                                                                      *
*  variable list:                                                      *
*                                                                      *
*   nx,ny     - 2-D grid size (x,y)                                    *
*   x(nx,ny) - x and y position data from CFD calculations             *
*   y(nx,ny) -                                                         *
*   u(nx,ny) - x and y velocity components from CFD calculations       *
*   v(nx,ny) -                                                         *
*   dt        - computed time step size for path calculations          *
*   maxp      - maximum points allowed for a path                      *
*   np        - number of points calculated for a path                 *
*   px(maxp) - x and y positions of calculated points for a path       *
*   py(maxp) -                                                         *
*   pu        - x and y velocity calculated for a points on the path   *
*   pv        -                                                        *
*   xint      - x and y starting position                              *
*   yint      -                                                        *
*   nobj      - number of graphical objects created                    *
*   iflag     - equals zero if xint,yint is a valid location           *
*                                                                      *
*  subroutines called:                                                 *
*                                                                      *
*   getdat  - get data from CFD data file                              *
*   dtime   - compute time step size                                   *
*   edges   - compute edges of flow field (max & min coordinates)      *
*   igraph  - initialize graphics                                      *
*   defobj  - define graphical objects (particle traces, backgrounds)  *
*   backgr  - draw background images                                   *
*   heading - dispaly heading text                                     *
*   pinput  - process input (starting location) from left mouse button *
*   trace   - calculate a particle trace                               *
*   modobj  - modify graphical objects by adding the particle trace    *
*                                                                      *
************************************************************************

#     include "fgl.h"
#     include "fdevice.h"

      integer nx,ny,np,maxp,nobj,iflag
      parameter (nx=46,ny=24,maxp=1500,nobj=75)
      real x(nx,ny),y(nx,ny),u(nx,ny),v(nx,ny)
      real px(maxp),py(maxp),dt,xint,yint

      call getdat(nx,ny,x,y,u,v)
      call dtime(nx,ny,x,y,u,v,dt)
      call edges(nx,ny,x,y)
      call igraph
      call defobj(nobj,nx,ny,x,y,u,v)
      call backgr(nobj)
      call headng(nx,ny,dt)
      call qdevic(LEFTMO)
      call qreset
      call writem(7)
      do while (1)

* begin animation loop

        do while((qtest()).eq.0)
          do 100 i = 1,nobj
            call color(BLACK)
            call clear
            call callob(i)
            call swapbu
 100      continue
        end do

* calculate a particle trace

        call pinput(xint,yint,iflag)
        if (iflag.eq.0) then
          call trace(nx,ny,x,y,u,v,xint,yint,px,py,maxp,np,dt,nflag)
          if (nflag.eq.0) then
            print *,'not added to graphical objects'
          else
            call modobj(nobj,px,py,maxp,np)
          endif
        endif
        call qreset
      end do

      end
*
*
*
      subroutine getdat(nx,ny,x,y,u,v)

* input CFD position and velocity data

      integer nx,ny,nx1,ny1
      real x(nx,ny),y(nx,ny),u(nx,ny),v(nx,ny)

      open(11,form='unformatted',file='mint.bin')
      read(11) ny1,nx1
      if (nx1.ne.nx.or.ny1.ne.ny) then
```

```
          print *,'wrong grid size !'
          stop
        endif
        read(11)((y(i,j),j=1,ny),i=1,nx)
        read(11)((x(i,j),j=1,ny),i=1,nx)
        read(11)((v(i,j),j=1,ny),i=1,nx)
        read(11)((u(i,j),j=1,ny),i=1,nx)

        return
        end
*
*
*
        subroutine pinput(xint,yint,iflag)

* process input (xint,yint) from queued device (left mouse button)

#       include "fgl.h"
#       include "fdevice.h"
#       include "fget.h"

        integer sx,sy,iflag
        integer lft,rght,bttm,tp
        real xint,yint,xmax,xmin,ymax,ymin
        real xlow,xup,ylow,yup,xfact,yfact,dx,dy
        common/bnd/xmax,xmin,ymax,ymin

        iflag = 0
        if ((qtest()).eq.LEFTMO) then

* read screen pixel coordinates and convert to world coordinates

          sx = getval(MOUSEX)
          sy = getval(MOUSEY)
          dx = (xmax-xmin)
          dy = (ymax-ymin)
          xup = xmax + .1*dx
          xlow = xmin - .1*dx
          yup = ymax + .3*dy
          ylow = ymin -dy -.3*dy
          call getori(lft,bttm)
          call getsiz(rght,tp)
          rght = lft + rght
          tp = bttm + tp
          xfact = (real(sx-lft))/(real(rght-lft))
          yfact = (real(sy-bttm))/(real(tp-bttm))
          xint = xlow + (xup-xlow)*xfact
          yint = ylow + (yup-ylow)*yfact
          if (xint.lt.xmin.or.xint.gt.xmax.or.
     &      yint.lt.ymin.or.yint.gt.ymax) then
            print *,' invalid cursor position '
            iflag = 1
          endif
        else
          iflag = 1
```

```
      endif

      return
      end
*
*
*
      subroutine dtime(nx,ny,x,y,u,v,dt)

* compute time step size as some multiple (dtmult) of the time required
* for the fastest particle to cross its own CFD grid cell

      integer nx,ny
      real x(nx,ny),y(nx,ny),u(nx,ny),v(nx,ny),dt,dtmult

      dtmult = 100
      dt = 1.e+6
      do 10 i = 2,nx-1
        do 10 j = 2,ny-1
          dt = amin1(dt,dtmult*(x(i+1,j)-x(i,j))/(abs(u(i,j))+1.e-10))
          dt = amin1(dt,dtmult*(y(i+1,j)-y(i,j))/(abs(v(i,j))+1.e-10))
 10   continue

      return
      end
*
*
*
      subroutine edges(nx,ny,x,y)

* compute the edges (max and min coordinates) of the flow field geometry

      integer nx,ny
      real x(nx,ny),y(nx,ny),xmax,xmin,ymax,ymin
      common/bnd/xmax,xmin,ymax,ymin

      xmin=x(1,1)
      xmax=x(1,1)
      ymin=y(1,1)
      ymax=y(1,1)
      do 70 i=1,nx
        do 70 j=1,ny
          if (x(i,j).lt.xmin) xmin=x(i,j)
          if (x(i,j).gt.xmax) xmax=x(i,j)
          if (y(i,j).lt.ymin) ymin=y(i,j)
          if (y(i,j).gt.ymin) ymax=y(i,j)
 70   continue

      return
      end
*
*
*
      subroutine igraph
```

```
* initialize graphics window

#     include "fgl.h"
#     include "fdevice.h"

      real xmax,xmin,ymax,ymin,dx,dy,xup,xlow,yup,ylow
      common/bnd/xmax,xmin,ymax,ymin

      dx = (xmax-xmin)
      dy = (ymax-ymin)
      xup = xmax + .1*dx
      xlow = xmin - .1*dx
      yup = ymax + .3*dy
      ylow = ymin -dy -.3*dy
      call foregr
      call prefpo(0,640,0,512)
      iwop = winope('Particle Trace',14)
      call winpop
      call concav(.true.)
      call double
      call gconfi
      call ortho2(xlow,xup,ylow,yup)
      call swapin(3)

      return
      end
*
*
*
      subroutine headng(nx,ny,dt)

* display heading (textual info, legends, etc..) for the animation window

#     include "fgl.h"
#     include "fdevice.h"

      integer nx,ny,sx,sy
      integer lft,rght,bttm,tp
      real xmax,xmin,ymax,ymin,dt
      common/bnd/xmax,xmin,ymax,ymin

      do 100 k = 1,8
        call mapcol(655+k,100,100,100)
 100  continue
      dx = (xmax-xmin)
      dy = (ymax-ymin)
      xup = xmax + .1*dx
      xlow = xmin - .1*dx
      yup = ymax + .3*dy
      ylow = ymin -dy -.3*dy

      call frontb(.true.)
      call color(656)
      xp = xmin
      yp = ymin - .2*dy
      call cmov2(xp,yp)
      call charst('Animated particle trace #1                     ',46)
      yp = yp - (32./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('  CFD data file: mint.bin                      ',46)
      yp = yp - (24./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('    grid size: ( 46 , 24 )                     ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('    (xmin,xmax): ( 0.0 , .3556 ) m             ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('    (ymin,ymax): ( 0.0 , .04445 ) m            ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('    dt: .03673 m/s                             ',46)
      yp = yp - (24./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('To initiate a particle trace, place cursor    ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('in flow field, and press left mouse button.   ',46)
      call frontb(.false.)

      return
      end
*
*
*
      subroutine defobj(nobj,nx,ny,x,y,u,v)

* define graphical objects #1 to #nobj for the particle path storage,
* GRID for the grid, and REACT for the reactor boundries (where V = 0)

#     include "fgl.h"
#     include "fdevice.h"

      integer nobj,GRID,REACT,nx,ny
      real x(nx,ny),y(nx,ny),u(nx,ny),v(nx,ny),vert(2)

      GRID = nobj+1
      REACT = nobj+2

      do 100 k = 1,8
        call mapcol(639+k,10,10,10)
        call mapcol(647+k,0,0,155)
 100  continue
      call mapcol(641,255,0,0)

      do 150 j = 1,nobj
        call makeob(j)
          call color(RED)
          call linewi(2)
        call closeo(j)
 150  continue

      call makeob(GRID)
        call color(640)
        call linewi(1)
        do 250 i = 1,nx
          call bgnlin
            do 200 j = 1,ny
              vert(1) = x(i,j)
              vert(2) = y(i,j)
              call v2f(vert)
200         continue
          call endlin
250     continue
        do 350 j = 1,ny
          call bgnlin
            do 300 i = 1,nx
              vert(1) = x(i,j)
              vert(2) = y(i,j)
              call v2f(vert)
300         continue
          call endlin
350     continue
      call closeo(GRID)

      call makeob(REACT)
        call color(648)
        call linewi(3)
        do 400 j = 1,ny-1
         do 400 i = 1,nx-1
           flag1 = 0
           flag2 = 0
           flag3 = 0
           flag4 = 0
           flagt = 0
           if (u(i,j).eq.0.0.and.v(i,j).eq.0.0) then
             flag1 = 1
           endif
           if (u(i+1,j).eq.0.0.and.v(i+1,j).eq.0.0) then
             flag2 = 1
           endif
           if (u(i,j+1).eq.0.0.and.v(i,j+1).eq.0.0) then
             flag3 = 1
           endif
           if (u(i+1,j+1).eq.0.0.and.v(i+1,j+1).eq.0.0) then
             flag4 = 1
           endif
           flagt = flag1+flag2+flag3+flag4
           if (flagt.ge.2) then
*
*     draw solid surface
*
             if (flagt.eq.4) then
               call bgnpol
                 vert(1) = x(i,j)
              vert(2) = y(i,j)
              call v2f(vert)
              vert(1) = x(i+1,j)
              vert(2) = y(i+1,j)
              call v2f(vert)
              vert(1) = x(i+1,j+1)
              vert(2) = y(i+1,j+1)
              call v2f(vert)
              vert(1) = x(i,j+1)
              vert(2) = y(i,j+1)
              call v2f(vert)
            call endpol
          endif
*
*      draw solid walls
*
          if (flag1.eq.1.and.flag2.eq.1) then
            call bgnlin
              vert(1) = x(i,j)
              vert(2) = y(i,j)
              call v2f(vert)
              vert(1) = x(i+1,j)
              vert(2) = y(i+1,j)
              call v2f(vert)
            call endlin
          endif
          if (flag1.eq.1.and.flag3.eq.1) then
            call bgnlin
              vert(1) = x(i,j)
              vert(2) = y(i,j)
              call v2f(vert)
              vert(1) = x(i,j+1)
              vert(2) = y(i,j+1)
              call v2f(vert)
             call endlin
           endif
           if (flag4.eq.1.and.flag2.eq.1) then
             call bgnlin
               vert(1) = x(i+1,j+1)
               vert(2) = y(i+1,j+1)
               call v2f(vert)
               vert(1) = x(i+1,j)
               vert(2) = y(i+1,j)
               call v2f(vert)
             call endlin
           endif
           if (flag4.eq.1.and.flag3.eq.1) then
             call bgnlin
               vert(1) = x(i+1,j+1)
               vert(2) = y(i+1,j+1)
               call v2f(vert)
               vert(1) = x(i,j+1)
               vert(2) = y(i,j+1)
               call v2f(vert)
             call endlin
```

```
              endif
            endif
 400      continue
        call closeo(REACT)

        return
        end
*
*
*
        subroutine backgr(nobj)

* draw background images for the animation sequences only once

#       include "fgl.h"
#       include "fdevice.h"

        integer nobj,GRID,REACT
        real xmax,xmin,ymax,ymin
        common/bnd/xmax,xmin,ymax,ymin

        GRID = nobj+1
        REACT = nobj+2
        call frontb(.true.)
        call color(BLACK)
        call clear
        call callob(GRID)
        call callob(REACT)
        call frontb(.false.)

        return
        end
*
*
*
        subroutine modobj(nobj,px,py,maxp,np)

* modify the graphical objects by adding the path recently defined

#       include "fgl.h"
#       include "fdevice.h"

        integer nobj,maxp,np,bsplin
        real px(maxp),py(maxp),xpt,ypt,geom(3,4)
        real xmax,xmin,ymax,ymin,rad,msix,asix,tthr,bspmat(4,4)
        common/bnd/xmax,xmin,ymax,ymin
        parameter (bsplin=3,msix=-1.0/6.0,asix=1.0/6.0,tthr=2.0/3.0)

        data bspmat/ msix,    .5,   -.5,   asix,
       +                .5,  -1.0,    .5,    0.0,
       +               -.5,   0.0,    .5,    0.0,
       +              asix,  tthr,  asix,    0.0/

        npm3 = np-3
        call defbas(bsplin,bspmat)
```

```
      call curveb(bsplin)

* represent particles as B-splines through 4 consecutive time steps

* for a given path, draw particles every nobj time step
* and increment 1 timestep for every graphical object

      do 100 i = 1,nobj
        do 200 j = i,npm3,nobj
          ic = 0
          do 300 k = j,j+3
            ic = ic+1
            geom(1,ic) = px(k)
            geom(2,ic) = py(k)
            geom(3,ic) = 0.0
 300      continue
          call editob(i)
            call crv(geom)
          call closeo(i)
 200    continue

* eliminate disappearing particles to avoid blinking during animation

        if (i.gt.npm3) then
          j = i-((i-1)/npm3)*npm3
          ic = 0
          do 400 k = j,j+3
            ic = ic+1
            geom(1,ic) = px(k)
            geom(2,ic) = py(k)
            geom(3,ic) = 0.0
 400      continue
          call editob(i)
            call crv(geom)
          call closeo(i)
        endif

 100  continue

      return
      end
*
*
*
      subroutine trace(nx,ny,x,y,u,v,xint,yint,px,py,maxp,np,dt,nflag)
**********************************************************************
*                                                                    *
* trace particle path from starting point (xint,yint) and store in px,py*
* using a Huen predictor/corrector time stepping method and a Lagrange *
* shape function (linear)                                            *
*                                                                    *
* Variable list:                                                     *
*                                                                    *
* xint,yint - starting location of particle trace                    *
* px,py     - coordinates storing particle trace                     *
* pu,pv      - x,y components of particle velocity at current location  *
* pui,pvi    -    "                              " at intermediate location *
* dt         - time step inetrval                                        *
* maxp       - maximum number of allowed time steps                     *
* np         - total # of timesteps for a particle trace                *
* no         - equals 0 until particle trace is complete                *
* nstag      - stagnation counter for stagnation stopping criteria      *
* stot       - total path length                                        *
* ncross     - crossing counter for recirculating flow stopping criteria *
* epsil      - minimum limit set on particle length                     *
* nflag      - equals 0 if particle trace has insignificant length      *
*                                                                       *
* Subroutines called:                                                   *
*                                                                       *
* pvel   - computes particle velocity from nodal velocities             *
* chkbnd - check boundary crossing flow                                 *
* pfin   - check stopping criteria to determine if path is finished     *
*                                                                       *
*************************************************************************

      integer np,nx,ny,maxp,no,nstag,ncross
      real px(maxp),py(maxp),pu1,pv1,pu2,pv2,stot,a1,b1
      real x(nx,ny),y(nx,ny),u(nx,ny),v(nx,ny),dt,epsil                  *
      common/ends/no,nstag,ncross,stot,a1,b1

      px(1) = xint
      py(1) = yint
      np = 1
      no = 0
      nstag = 0
      stot = 0.0
      ncross = 1
      epsil = 1.e-6
      nflag = 1
      do while (no.eq.0)
         np = np + 1
         call pvel(nx,ny,x,y,u,v,px,py,maxp,pu,pv,np)
         px(np) = px(np-1) + pu*dt
         py(np) = py(np-1) + pv*dt
         call ckbnd(px,py,maxp,np)
         np = np+1
         call pvel(nx,ny,x,y,u,v,px,py,maxp,pui,pvi,np)
         np = np-1
         px(np) = px(np-1) + ((pu+pui)/2.0)*dt
         py(np) = py(np-1) + ((pv+pvi)/2.0)*dt
         call ckbnd(px,py,maxp,np)
         call pfin(px,py,maxp,np)
      end do
      if (stot.lt.epsil) nflag = 0

      return
      end
*
*
*
```

```
      subroutine pvel(nx,ny,x,y,u,v,px,py,maxp,pu,pv,np)

* compute particle velocity from nodal velocities based on particle
* location within the coordinate grid
*
* pu,pv - x,y components of particle velocity
* md    - node identifier if enclosed by four nodes (lower,left node)
* od    - node identifier if on a node
*
* Subroutines called:
*
* ploc  - determine particle location with respect to nodal (i,j) indices
* mvel  - compute particle velocity using appropriate nodal interpolation
*

      integer nx,ny,maxp,np,md,od
      real px(maxp),py(maxp),pu,pv
      real x(nx,ny),y(nx,ny),u(nx,ny),v(nx,ny)

      call ploc(nx,ny,x,y,px,py,maxp,np,od,md)
      if (od.ne.0) then

* particle is on a node, set particle velocity to nodal velocity

        ip = (od-1)/ny + 1
        jp = od - (ip-1)*ny
        pu = u(ip,jp)
        pv = v(ip,jp)
      else if (md.ne.0) then

* particle between four nodes, interploate particle velocity

        call mvel(nx,ny,x,y,u,v,md,px,py,maxp,np,pu,pv)
      else
        print *,'particle out of grid'
        stop
      endif

      return
      end
*
*
*
      subroutine ploc(nx,ny,x,y,px,py,maxp,np,od,md)

* determine particle location (between nodes or on a node)

* md - node identifier if enclosed by four nodes (lower,left node)
* od - node identifier if on a node

      integer nx,ny,maxp,np,md,od
      real x(nx,ny),y(nx,ny),px(maxp),py(maxp)

      od = 0
      md = 0
      do 40 i=1,nx-1
        do 40 j=1,ny-1
          if (px(np-1).ge.x(i,j).and.px(np-1).le.x(i+1,j).
     &    and.py(np-1).ge.y(i,j).and.py(np-1).le.y(i,j+1)) then
            md=(i-1)*ny + j
            goto 25
          endif
  40  continue
      do 50 i=1,nx
        do 50 j=1,ny
          if (px(np-1).eq.x(i,j).and.py(np-1).eq.y(i,j)) then
            od = (i-1)*ny + j
            goto 25
          endif
  50  continue
  25  continue

      return
      end
*
*
*
      subroutine mvel(nx,ny,x,y,u,v,md,px,py,maxp,np,pu,pv)

* Lagrange shape function (linear) interploation of particles velocity
*
* Variables:
*
* xc,yc - x,y center of grid cell which encloses the particle
* b,a   - 1/2 of the width and heigth of the grid cell
* xx,yy - x,y displacement of particle from center of grid cell
* c1,c2 - coordinates of particle in natural coordinate system
* e(4)  - Lagrange shape functions

      integer nx,ny,maxp,np,md
      real x(nx,ny),y(nx,ny),u(nx,ny),v(nx,ny)
      real px(maxp),py(maxp),pu,pv
      real xc,yc,xx,yy,b,a,c1,c2,e(4)

      i = (md-1)/ny + 1
      j = md - (i-1)*ny

      xc = (x(i+1,j) + x(i,j))/2.0
      yc = (y(i,j+1) + y(i,j))/2.0
      b = (x(i+1,j) - x(i,j))/2.0
      a = (y(i,j+1) - y(i,j))/2.0
      xx = px(np-1) - xc
      yy = py(np-1) - yc
      c1 = xx/b
      c2 = yy/a

      e(1) = .25*(1-c1)*(1-c2)
      e(2) = .25*(1+c1)*(1-c2)
      e(3) = .25*(1+c1)*(1+c2)
      e(4) = .25*(1-c1)*(1+c2)
      pu=e(1)*u(i,j)+e(2)*u(i+1,j)+e(3)*u(i+1,j+1)+e(4)*u(i,j+1)
      pv=e(1)*v(i,j)+e(2)*v(i+1,j)+e(3)*v(i+1,j+1)+e(4)*v(i,j+1)

      return
      end
*
*
*
      subroutine ckbnd(px,py,maxp,np)

* check that particle does not cross grid boundry

      integer maxp,np
      common/bnd/xmax,xmin,ymax,ymin
      real px(maxp),py(maxp)

      if (px(np).lt.xmin) then
        px(np) = (px(np-1) - xmin)/2.0
      endif
      if (px(np).gt.xmax) then
        px(np) = px(np-1) + (xmax - px(np-1))/2.0
      endif
      if (py(np).lt.ymin) then
        py(np) = (py(np-1) - ymin)/2.0
      endif
      if (py(np).gt.ymax) then
        py(np) = py(np-1) + (ymax - py(np-1))/2.0
      endif

      return
      end
*
*
*
      subroutine pfin(px,py,maxp,np)

* determine when particle path is finished, no = 1 when complete

      integer maxp,np,no,nstag,ncross
      real px(maxp),py(maxp),s,stot
      real a,b,a1,b1,det(3),xi,yi,xlow,ylow,xhi,yhi
      character*25 end
      common/ends/no,nstag,ncross,stot,a1,b1

* Stopping criteria: particle stagnation

*   if particle takes excessive number of insignificant
*   timesteps, then end path calculation

* s     - length of current timestep
* stot  - total path length
* epsil - insignificant timestep
* nstag - stagnation counter
```

```
      s = (px(np)-px(np-1))**2 + (py(np)-py(np-1))**2
      s = sqrt(s)
      stot = stot+s
      epsil = 1.e-10
      if (s.lt.epsil) then
        nstag = nstag + 1
        if (nstag.eq.50) then
          end = 'particle stagnated'
c          print 10,end,px(1),py(1),stot,np
          no = 1
          goto 25
        endif
      endif

* Stopping criteria: maximum particle limit

      if (np.eq.maxp) then
        end = 'max particle limit'
c        print 10,end,px(1),py(1),stot,np
        no = 1
        goto 25
      endif

* Stopping criteria: recirculating flow

*   if particle path crosses line perpindular to first time step
*   more than twice, then flow has recirculated

* a1      - slope of line 1, perpindicular to first time step
* b1      - y-intercept of line 1, perpindicular to first time step
* a       - slope of line segment i, for current time step
* b       - y-intercept of line segment i, for current time step
* xi,yi   - intersection of line 1 and line i
* xhi,yhi - box enclosing line segment i
* xlow,ylow
* det()   - determinants computed for Cramer's rule
* ncross  - crossing counter determined by line intersections

      if (np.eq.2) then
        a1=(px(1)-px(2))/(py(2)-py(1))
        b1=py(1)-px(1)*a1
        no = 0
        goto 25
      endif
      a=(py(np)-py(np-1))/(px(np)-px(np-1))
      b=py(np-1)-px(np-1)*a
      det(1) = a1 - a
      if (det(1).ne.0.0) then
        det(2) = a1*b - a*b1
        det(3) = b - b1
        yi = det(2)/det(1)
        xi = det(3)/det(1)
        xlow = amin1(px(np),px(np-1))
        ylow = amin1(py(np),py(np-1))
        xhi = amax1(px(np),px(np-1))
        yhi = amax1(py(np),py(np-1))
        if (xl.ge.xlow.and.xl.le.xhi.and.
     &  yl.ge.ylow.and.yl.le.yhi) then
          ncross = ncross + 1
          if (ncross.eq.3) then
            no = 1
            np = np - 1
            end = 'path closed'
c            print 10,end,px(1),py(1),stot,np
            nflag = 1
          endif
        else
          no = 0
        endif
      else
         no = 0
      endif

 25   continue
 10   format(1x,a20,3(f6.4,2x),i6)
      return
      end
*
```

Appendix C

Modified Subroutines for other Apt's

```
      program apt2
**************************************************************************
*                                                                        *
*                                                          4/15/92       *
* Ted Blackmon, GFW                                                      *
* Applied Researh Lab                                                    *
* Penn State University                                                  *
*                                                                        *
* Animated particle trace for a 2-D, steady state, single phase,         *
* recirculating flow in a closed combustor of a liquid metal fuel        *
*                                                                        *
*                                                                        *
*   -- Starting location of particles through input deck in program;     *
*        starting locations evenly distributed throughout flow field     *
*   -- Linear shape function velocity approximation                      *
*   -- Huen predictor/corrector time stepping method                     *
*   -- Background is the coordinate grid                                 *
*                                                                        *
*  variable list:                                                        *
*                                                                        *
*   nx,ny      - 2-D grid size (x,y)                                     *
*   x(nx,ny) - x and y position data from CFD calculations               *
*   y(nx,ny) -                                                           *
*   u(nx,ny) - x and y velocity components from CFD calculations         *
*   v(nx,ny) -                                                           *
*   dt         - computed time step size for path calculations           *
*   maxp       - maximum points allowed for a path                       *
*   np         - number of points calculated for a path                  *
*   px(maxp) - x and y positions of calculated points for a path         *
*   py(maxp) -                                                           *
*   pu         - x and y velocity calculated for a points on the path    *
*   pv         -                                                         *
*   xint       - x and y starting position                               *
*   yint       -                                                         *
*   nobj       - number of graphical objects created                     *
*   iflag      - equals zero if xint,yint is a valid location            *
*                                                                        *
*  subroutines called:                                                   *
*                                                                        *
*   getdat   - get data from CFD data file                               *
*   dtime    - compute time step size                                    *
*   edges    - compute edges of flow field (max & min coordinates)       *
*   igraph   - initialize graphics                                       *
*   defobj   - define graphical objects (particle traces, backgrounds)   *
*   backgr   - draw background images                                    *
*   heading  - dispaly heading text                                      *
*   pinput   - process input (starting location) from left mouse button  *
*   trace    - calculate a particle trace                                *
*   modobj   - modify graphical objects by adding the particle trace     *
*                                                                        *
**************************************************************************

#     include "fgl.h"
#     include "fdevice.h"

      integer nx,ny,np,maxp,nobj,iflag
      parameter (nx=46,ny=24,maxp=200,nobj=100)
      real x(nx,ny),y(nx,ny),u(nx,ny),v(nx,ny)
      real px(maxp),py(maxp),dt,xint,yint
      real xmax,xmin,ymax,ymin
      common/bnd/xmax,xmin,ymax,ymin

      call getdat(nx,ny,x,y,u,v)
      call dtime(nx,ny,x,y,u,v,dt)
      call edges(nx,ny,x,y)
      call igraph
      call defobj(nobj,nx,ny,x,y,u,v)
      call backgr(nobj)
      call headng(nx,ny,dt)

* calculate particle traces

      ntrace = 0
      dy = (ymax-ymin)/10
      dx = (xmax-xmin)/50
      do 25 xint = xmin+dx,xmax-dx,dx
       do 50 yint = ymin+dy,ymax-dy,dy
         call trace(nx,ny,x,y,u,v,xint,yint,px,py,maxp,np,dt,nflag)
         if (nflag.eq.0) then
            print *,'not added to graphical objects'
            nflag = nflag
         else
           ntrace = ntrace + 1
           call modobj(nobj,px,py,maxp,np)
         endif
 50    continue
 25   continue

* animation loop

      call writem(7)
      do while (1)
       do 100 i = 1,nobj
        call color(BLACK)
        call clear
        call callob(i)
        call swapbu
 100   continue
      end do
      call qreset

      end
*
*
*
      subroutine headng(nx,ny,dt)

* display heading and message to start new particle trace

#     include "fgl.h"
#     include "fdevice.h"
```

```
      integer nx,ny
      real xmax,xmin,ymax,ymin,dt
      common/bnd/xmax,xmin,ymax,ymin

      do 100 k = 1,8
        call mapcol(655+k,100,100,100)
 100  continue
      dx = (xmax-xmin)
      dy = (ymax-ymin)
      xup = xmax + .1*dx
      xlow = xmin - .1*dx
      yup = ymax + .3*dy
      ylow = ymin -dy -.3*dy

      call frontb(.true.)
      call color(656)
      xp = xmin
      yp = ymin - .2*dy
      call cmov2(xp,yp)
      call charst('Animated particle trace #3                    ',46)
      yp = yp - (32./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('  CFD data file: mint.bin                     ',46)
      yp = yp - (24./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('   grid size: ( 46 , 24 )                     ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('   (xmin,xmax): ( 0.0 , .3556 ) m             ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('   (ymin,ymax): ( 0.0 , .04445 ) m            ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('   dt: .007347 m/s                            ',46)
      yp = yp - (24./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('Numerous particle traces initiated within source',49)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('source code.  Total # of particle traces: 508 ',46)
      call frontb(.false.)

      return
      end
*
*
*
```

```
      program apt3
************************************************************************
*                                                                      *
* Ted Blackmon, GFW                                          4/15/92  *
* Applied Researh Lab                                                  *
* Penn State University                                                *
*                                                                      *
* Animated particle trace for a 2-D, steady state, single phase,      *
* recirculating flow in a closed combustor of a liquid metal fuel     *
*                                                                      *
*                                                                      *
*   -- Starting location of particles through input deck within source *
*      code                                                            *
*   -- 98 particle traces for an initial column of starting locations  *
*      to produce a timeline                                           *
*   -- Linear shape function velocity approximation                    *
*   -- Huen predictor/corrector time stepping method                   *
*   -- Background is the coordinate grid                               *
*                                                                      *
*  variable list:                                                      *
*                                                                      *
*   nx,ny     - 2-D grid size (x,y)                                    *
*   x(nx,ny)  - x and y position data from CFD calculations            *
*   y(nx,ny)  -                                                        *
*   u(nx,ny)  - x and y velocity components from CFD calculations      *
*   v(nx,ny)  -                                                        *
*   dt        - computed time step size for path calculations          *
*   maxp      - maximum points allowed for a path                      *
*   np        - number of points calculated for a path                 *
*   px(maxp)  - x and y positions of calculated points for a path      *
*   py(maxp)  -                                                        *
*   pu        - x and y velocity calculated for a points on the path   *
*   pv        -                                                        *
*   xint      - x and y starting position                              *
*   yint      -                                                        *
*   nobj      - number of graphical objects created                    *
*   iflag     - equals zero if xint,yint is a valid location           *
*                                                                      *
*  subroutines called:                                                 *
*                                                                      *
*   getdat  - get data from CFD data file                              *
*   dtime   - compute time step size                                   *
*   edges   - compute edges of flow field (max & min coordinates)      *
*   igraph  - initialize graphics                                      *
*   defobj  - define graphical objects (particle traces, backgrounds)  *
*   backgr  - draw background images                                   *
*   heading - display heading text                                     *
*   trace   - calculate a particle trace                               *
*   modobj  - modify graphical objects by adding the particle trace    *
*                                                                      *
************************************************************************

#     include "fgl.h"
#     include "fdevice.h"
```

```
      program apt5
***********************************************************************
*                                                                     *
* Ted Blackmon, GFW                                          4/15/92  *
* Applied Research Lab                                                *
* Penn State University                                               *
*                                                                     *
* Animated particle trace for a 2-D, steady state, multiphase,        *
* recirculating flow in a closed combustor of a liquid metal fuel     *
*                                                                     *
*                                                                     *
*   -- Starting location of particles input within source code        *
*   -- Linear shape function velocity approximation                   *
*   -- Huen predictor/corrector time stepping method                  *
*   -- Mixture velocity of the three phases to push particles         *
*   -- Background is false coloring of numerical x-ray attenuation    *
*                                                                     *
*  variable list:                                                     *
*                                                                     *
*   nx,ny    - 2-D grid size (x,y)                                    *
*   x(nx,ny) - x and y position data from CFD calculations            *
*   y(nx,ny) -                                                        *
*   u(nx,ny) - x and y velocity components from CFD calculations      *
*   v(nx,ny) -                                                        *
*   den      - mixture density                                        *
*   gray     - numerical x-ray attenuation levels                     *
*   dt       - computed time step size for path calculations          *
*   maxp     - maximum points allowed for a path                      *
*   np       - number of points calculated for a path                 *
*   px(maxp) - x and y positions of calculated points for a path      *
*   py(maxp) -                                                        *
*   pu       - x and y velocity calculated for a points on the path   *
*   pv       -                                                        *
*   xint     - x and y starting position                              *
*   yint     -                                                        *
*   nobj     - number of graphical objects created                    *
*   iflag    - equals zero if xint,yint is a valid location           *
*                                                                     *
*  subroutines called:                                                *
*                                                                     *
*   getdat  - get data from CFD data file                             *
*   dtime   - compute time step size                                  *
*   edges   - compute edges of flow field (max & min coordinates)     *
*   igraph  - initialize graphics                                     *
*   defobj  - define graphical objects (particle traces, backgrounds) *
*   backgr  - draw background images                                  *
*   headng  - display heading text                                    *
*   trace   - calculate a particle trace                              *
*   modobj  - modify graphical objects by adding the particle trace   *
*                                                                     *
***********************************************************************

#     include "fgl.h"
#     include "fdevice.h"
```

```
      integer nx,ny,np,maxp,nobj,iflag
      parameter (nx=29,ny=21,maxp=200,nobj=100)
      real x(nx,ny),y(nx,ny),u(nx,ny),v(nx,ny),den(nx,ny),gray(nx,ny)
      real px(maxp),py(maxp),dt,xint,yint
      real xmax,xmin,ymax,ymin
      common/bnd/xmax,xmin,ymax,ymin

      call getdat(nx,ny,x,y,u,v,den,gray)
      call dtime(nx,ny,x,y,u,v,dt)
      call edges(nx,ny,x,y)
      call igraph
      call defobj(nobj,nx,ny,x,y,u,v,gray)
      call dinit(nobj)
      call headng(nx,ny,dt)

* calculate particle traces

      ntrace = 0
      dy = (ymax-ymin)/15
      dx = (xmax-xmin)/25
      do 25 xint = xmin+dx,xmax-dx,dx
       do 50 yint = ymin+dy,ymax-dy,dy
         call trace(nx,ny,x,y,u,v,xint,yint,px,py,maxp,np,dt,nflag)
         if (nflag.eq.0) then
            print *,'not added to graphical objects'
            nflag = nflag
         else
           ntrace = ntrace + 1
           call modobj(nobj,px,py,maxp,np)
         endif
 50    continue
 25   continue

* begin animation loop

      do 75 k = 1,256
        call mapcol(257+2*k,255,255,255)
 75   continue
      call mapcol(1,255,255,255)
      call writem(1)
      do while (1)
       do 100 i = 1,nobj
        call color(BLACK)
        call clear
        call callob(i)
        call swapbu
 100   continue
      end do
      call qreset

      end
*
*
*
      subroutine getdat(nx,ny,x,y,u,v,den,gray)
```

```
* Input CFD position and velocity data

      integer nx,ny,ni,nj
      real x(nx,ny),y(nx,ny),u(nx,ny),v(nx,ny)
      real den(nx,ny),gray(nx,ny),fsmach,alpha,re,time

      call copen(12,'u2dxray.bin','r',istat)
      call copen(14,'x2dxray.bin','r',istat)

      call cread(12,4,ni,istat)
      call cread(12,4,nj,istat)
      print *,'ni =',ni
      print *,'nj =',nj
      if (ni.ne.nx.or.nj.ne.ny) then
         print *,'wrong grid size !'
         stop
      endif
      call cread(12,4,fsmach,istat)
      call cread(12,4,alpha,istat)
      call cread(12,4,re,istat)
      call cread(12,4,time,istat)
      do 10 j = 1,ny
        do 10 i = 1,nx
          call cread(12,4,den(i,j),istat)
 10   continue
      do 20 j = 1,ny
        do 20 i = 1,nx
          call cread(12,4,u(i,j),istat)
 20   continue
      do 30 j = 1,ny
        do 30 i = 1,nx
          call cread(12,4,v(i,j),istat)
 30   continue
      do 40 j = 1,ny
        do 40 i = 1,nx
          call cread(12,4,gray(i,j),istat)
 40   continue
      call cread(14,4,ni,istat)
      call cread(14,4,nj,istat)
      if (ni.ne.nx.or.nj.ne.ny) then
         print *,'wrong grid size !'
         stop
      endif
*      call cread(14,4,fsmach,istat)
*      call cread(14,4,alpha,istat)
*      call cread(14,4,re,istat)
*      call cread(14,4,time,istat)
      do 50 j = 1,ny
        do 50 i = 1,nx
          call cread(14,4,x(i,j),istat)
 50   continue
      do 60 j = 1,ny
        do 60 i = 1,nx
          call cread(14,4,y(i,j),istat)
 60   continue

      return
      end
*
*
*
      subroutine headng(nx,ny,dt)

* display heading and message to start new particle trace

#     include "fgl.h"
#     include "fdevice.h"

      integer nx,ny,sx,sy
      real xmax,xmin,ymax,ymin,dt,vert(2)
      common/bnd/xmax,xmin,ymax,ymin

      call mapcol(770,125,125,125)
      call mapcol(771,125,125,125)

      dx = (xmax-xmin)
      dy = (ymax-ymin)
      xup = xmax + .1*dx
      xlow = xmin - .1*dx
      yup = ymax + .35*dy
      ylow = ymin -dy -.3*dy

      call frontb(.true.)
      call color(770)
      xp = xmin
      yp = ymin - .2*dy
      call cmov2(xp,yp)
      call charst('Animated particle trace #5                    ',46)
      yp = yp - (32./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('  CFD data file: x2dxray.bin                  ',46)
      yp = yp - (24./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('    grid size: ( 29 , 23 )                    ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('    (xmin,xmax): ( 0.0 , 1.049 ) m            ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('    (ymin,ymax): ( 0.0 , .3937 ) m            ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('    dt: 4.70e-4 m/s                           ',46)
      yp = yp - (24./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('Uses mixture velocity for particle traces     ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('Background: false coloring of x-ray attenuation',47)
      yp = yp - (24./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('Particle traces initiated through input deck  ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('within source code. Total # particle traces: 504',48)

* draw gravitational vector

      call bgnlin
        vert(1) = xmax - .05*dx
        vert(2) = ymin - .1*dy
        call v2f(vert)
        vert(1) = xmax
        vert(2) = ymin - .1*dy - .05*dx
        call v2f(vert)
        vert(1) = xmax
        vert(2) = ymin - .1*dy - .03*dx
        call v2f(vert)
      call endlin
      call bgnlin
        vert(1) = xmax
        vert(2) = ymin - .1*dy - .05*dx
        call v2f(vert)
        vert(1) = xmax - .02*dx
        vert(2) = ymin - .1*dy - .05*dx
        call v2f(vert)
      call endlin
      xp = xmax
      yp = ymin - .1*dy
      call cmov2(xp,yp)
      call charst('g',1)

* draw color scale legend

      xp = xmax - .15*dx
      yp = ymin - .2*dy - .09*dx
      call cmov2(xp,yp)
      call charst('Attenuation',11)

      xp = xmax - .1*dx
      yp = yp - (18./512.)*(yup - ylow)
      nscale = 258 + nint(1.0*511.0)
      call color(nscale)
      call cmov2(xp,yp)
      call charst('1.00',4)

      yp = yp - (12./512.)*(yup - ylow)
      nscale = 258 + nint(.90*511.0)
      call color(nscale)
      call cmov2(xp,yp)
      call charst('0.90',4)

      yp = yp - (12./512.)*(yup - ylow)
      nscale = 258 + nint(.80*511.0)
      call color(nscale)
      call cmov2(xp,yp)
      call charst('0.80',4)

      yp = yp - (12./512.)*(yup - ylow)
      nscale = 258 + nint(.70*511.0)
      call color(nscale)
      call cmov2(xp,yp)
      call charst('0.70',4)

      yp = yp - (12./512.)*(yup - ylow)
      nscale = 258 + nint(.60*511.0)
      call color(nscale)
      call cmov2(xp,yp)
      call charst('0.60',4)

      yp = yp - (12./512.)*(yup - ylow)
      nscale = 258 + nint(.50*511.0)
      call color(nscale)
      call cmov2(xp,yp)
      call charst('0.50',4)

      yp = yp - (12./512.)*(yup - ylow)
      nscale = 258 + nint(.40*511.0)
      call color(nscale)
      call cmov2(xp,yp)
      call charst('0.40',4)

      yp = yp - (12./512.)*(yup - ylow)
      nscale = 258 + nint(.30*511.0)
      call color(nscale)
      call cmov2(xp,yp)
      call charst('0.30',4)

      yp = yp - (12./512.)*(yup - ylow)
      nscale = 258 + nint(.20*511.0)
      call color(nscale)
      call cmov2(xp,yp)
      call charst('0.20',4)

      yp = yp - (12./512.)*(yup - ylow)
      nscale = 258 + nint(.10*511.0)
      call color(nscale)
      call cmov2(xp,yp)
      call charst('0.10',4)

      yp = yp - (12./512.)*(yup - ylow)
      nscale = 258 + nint(.0*511.0)
      call color(nscale)
      call cmov2(xp,yp)
      call charst('0.00',4)

      call frontb(.false.)

      return
      end
*
*
*
      subroutine defobj(nobj,nx,ny,x,y,u,v,gray)

* define graphical objects #1 to #nobj for the paths,
* GRID for the grid, and REACT for the reactor boundries
* and ATTEN for the color contour of x-ray attenuation data

#     include "fgl.h"
#     include "fdevice.h"

      integer nobj,GRID,REACT,ATTEN,nx,ny
      real x(nx,ny),y(nx,ny),u(nx,ny),v(nx,ny),gray(nx,ny)
      real vert(2)

      GRID = nobj+1
      REACT = nobj+2
      ATTEN = nobj+3

      do 100 k = 1,2
        call mapcol(255+k,255,0,255)
 100  continue
      do 125 k = 1,128
        call mapcol(257+2*k-1,256-2*k,2*k-1,0)
        call mapcol(257+2*k,256-2*k,2*k-1,0)
        call mapcol(513+2*k-1,0,256-2*k,2*k-1)
        call mapcol(513+2*k,0,256-2*k,2*k-1)
 125  continue

      do 150 j = 1,nobj
        call makeob(j)
          call color(1)
          call linewi(2)
        call closeo(j)
 150  continue

      call makeob(GRID)
        call color(640)
        call linewi(1)
        do 250 i = 1,nx
          call bgnlin
            do 200 j = 1,ny
              vert(1) = x(i,j)
              vert(2) = y(i,j)
              call v2f(vert)
200         continue
          call endlin
250     continue
        do 350 j = 1,ny
          call bgnlin
            do 300 i = 1,nx
              vert(1) = x(i,j)
              vert(2) = y(i,j)
          call v2f(vert)
300     continue
       call endlin
350   continue
      call closeo(GRID)

      call makeob(REACT)
        call color(256)
        call linewi(3)
        do 400 j = 1,ny-1
         do 400 i = 1,nx-1
           flag1 = 0
           flag2 = 0
           flag3 = 0
           flag4 = 0
           flagt = 0
           if (u(i,j).eq.0.0.and.v(i,j).eq.0.0) then
             flag1 = 1
           endif
           if (u(i+1,j).eq.0.0.and.v(i+1,j).eq.0.0) then
             flag2 = 1
           endif
           if (u(i,j+1).eq.0.0.and.v(i,j+1).eq.0.0) then
             flag3 = 1
           endif
           if (u(i+1,j+1).eq.0.0.and.v(i+1,j+1).eq.0.0) then
             flag4 = 1
           endif
           flagt = flag1+flag2+flag3+flag4
           if (flagt.ge.2) then
*
*     draw solid surface
*
             if (flagt.eq.4) then
               call bgnpol
                 vert(1) = x(i,j)
                 vert(2) = y(i,j)
                 call v2f(vert)
                 vert(1) = x(i+1,j)
                 vert(2) = y(i+1,j)
                 call v2f(vert)
                 vert(1) = x(i+1,j+1)
                 vert(2) = y(i+1,j+1)
                 call v2f(vert)
                 vert(1) = x(i,j+1)
                 vert(2) = y(i,j+1)
                 call v2f(vert)
               call endpol
             endif
*
*      draw solid walls
*
             if (flag1.eq.1.and.flag2.eq.1) then
               call bgnlin
                 vert(1) = x(i,j)
```

```
              vert(2) = y(i,j)
              call v2f(vert)
              vert(1) = x(i+1,j)
              vert(2) = y(i+1,j)
              call v2f(vert)
            call endlin
          endif
          if (flag1.eq.1.and.flag3.eq.1) then
            call bgnlin
              vert(1) = x(i,j)
              vert(2) = y(i,j)
              call v2f(vert)
              vert(1) = x(i,j+1)
              vert(2) = y(i,j+1)
              call v2f(vert)
            call endlin
          endif
          if (flag4.eq.1.and.flag2.eq.1) then
            call bgnlin
              vert(1) = x(i+1,j+1)
              vert(2) = y(i+1,j+1)
              call v2f(vert)
              vert(1) = x(i+1,j)
              vert(2) = y(i+1,j)
              call v2f(vert)
            call endlin
          endif
          if (flag4.eq.1.and.flag3.eq.1) then
            call bgnlin
              vert(1) = x(i+1,j+1)
              vert(2) = y(i+1,j+1)
              call v2f(vert)
              vert(1) = x(i,j+1)
              vert(2) = y(i,j+1)
              call v2f(vert)
            call endlin
          endif
        endif
400   continue
      call closeo(REACT)

      call makeob(ATTEN)
        do 500 i = 2,nx-2
          do 500 j = 2,ny-2
            call bgnpol
              nscale = 258 + nint(gray(i,j)*511.0)
              call color(nscale)
              vert(1) = x(i,j)
              vert(2) = y(i,j)
              call v2f(vert)
              nscale = 258 + nint(gray(i+1,j)*511.0)
              call color(nscale)
              vert(1) = x(i+1,j)
              vert(2) = y(i+1,j)
              call v2f(vert)
```

```
                nscale = 258 + nint(gray(i+1,j+1)*511.0)
                call color(nscale)
                vert(1) = x(i+1,j+1)
                vert(2) = y(i+1,j+1)
                call v2f(vert)
                nscale = 258 + nint(gray(i,j+1)*511.0)
                call color(nscale)
                vert(1) = x(i,j+1)
                vert(2) = y(i,j+1)
                call v2f(vert)
              call endpol
 500      continue
        call closeo(ATTEN)

        return
        end
*
*
*
```

```
      program apt10
*********************************************************************
*                                                                   *
*                                                                   *
* Ted Blackmon, GFW                                       4/15/92   *
* Applied Researh Lab                                               *
* Penn State University                                             *
*                                                                   *
* Animated particle trace for a 2-D, steady state, single phase,    *
* recirculating flow in a closed combustor of a liquid metal fuel   *
*                                                                   *
*                                                                   *
*   -- Starting location of particles through input deck in program *
*   -- Multi-colored particles   GREEN - Fuel phase                 *
*                                BLUE  - Product phase              *
*                                RED   - Vapor phase                *
*   -- Particle color intensity controlled through volume fraction  *
*   -- Linear shape function velocity approximation                 *
*   -- Huen predictor/corrector time stepping method                *
*                                                                   *
*  variable list:                                                   *
*                                                                   *
*   nx,ny    - 2-D grid size (x,y)                                  *
*   nl       - total number of phases                               *
*   x(nx,ny) - x and y position data from CFD calculations          *
*   y(nx,ny) -                                                      *
*   u(nx,ny,nl) - velocity components for each phase                *
*   v(nx,ny,nl)                                                     *
*   uu(nx,ny)   - velocity components for particle trace calculations *
*   vv(nx,ny)                                                       *
*   den      - density values for each phase                        *
*   phi      - volume fraction values for each phase                *
*   phil     - volume fraction data for transparency control        *
*   dt       - computed time step size for path calculations        *
*   maxp     - maximum points allowed for a path                    *
*   np       - number of points calculated for a path               *
*   px(maxp) - x and y positions of calculated points for a path    *
*   py(maxp) -                                                      *
*   pu       - x and y velocity calculated for a points on the path *
*   pv       -                                                      *
*   pphi     - volume fraction values along the particle trace      *
*   xint     - x and y starting position                            *
*   yint     -                                                      *
*   nobj     - number of graphical objects created                  *
*   iflag    - equals zero if xint,yint is a valid location         *
*                                                                   *
*  subroutines called:                                              *
*                                                                   *
*   getdat  - get data from CFD data file                           *
*   dtime   - compute time step size                                *
*   edges   - compute edges of flow field (max & min coordinates)   *
*   igraph  - initialize graphics                                   *
*   defobj  - define graphical objects (particle traces, backgrounds) *
*   backgr  - draw background images                                *
*   heading - dispaly heading text                                  *
*   pinput  - process input (starting location) from left mouse button *
*    trace    - calculate a particle trace                                *
*    modobj   - modify graphical objects by adding the particle trace     *
*                                                                         *
***************************************************************************

#     include "fgl.h"
#     include "fdevice.h"

      integer nx,ny,np,maxp,nobj,iflag
      parameter (nx=29,ny=21,nl=3,maxp=200,nobj=100)
      real x(nx,ny),y(nx,ny),px(maxp),py(maxp),pphi(maxp),dt,xint,yint
      real uu(nx,ny),vv(nx,ny),u(nx,ny,nl),v(nx,ny,nl)
      real den(nx,ny,nl),phi(nx,ny,nl),phil(nx,ny)
      real xmax,xmin,ymax,ymin
      common/bnd/xmax,xmin,ymax,ymin

      BACKGR = nobj+3
      call getdat(nx,ny,nl,x,y,u,v,den,phi)
      call dtime(nx,ny,nl,x,y,u,v,dt)
      call edges(nx,ny,x,y)
      call igraph(nobj)
      call defobj(nobj,nx,ny,nl,x,y,u,v)
      call dinit(nobj)
      call headng(nx,ny,dt)
      call qdevic(LEFTMO)
      call qreset
      call blendf(BFSA,BFMSA)
      call afunct(0,AFNOTE)

* calculate particle traces

      print *,'dt =',dt
      pause
      dx = (xmax-xmin)/15
      dy = (ymax-ymin)/11
      do 200 xint = xmin+dx,xmax-dx,dx
        do 200 yint = ymin+dy,ymax-dy,dy
          do 200 l = 1,nl
            do 125 i = 1,nx
              do 125 j = 1,ny
                uu(i,j) = u(i,j,l)
                vv(i,j) = v(i,j,l)
                phil(i,j) = phi(i,j,l)
 125        continue
            call trace(nx,ny,x,y,uu,vv,xint,yint,px,py,maxp,np,dt,nflag)
            if (nflag.eq.0) then
              print *,'not added to graphical objects'
            else
              call transp(nx,ny,maxp,np,x,y,phil,px,py,pphi)
              call modobj(nx,ny,nobj,l,x,y,px,py,pphi,maxp,np)
            endif
 200  continue

* animation loop
```

```
      do while (1)
        do 100 i = 1,nobj
          call callob(BACKGR)
          call callob(i)
          call swapbu
 100    continue
      end do

      end
*
*
*
      subroutine getdat(nx,ny,nl,x,y,u,v,den,phi)

* input CFD position and velocity data

      integer nx,ny,ni,nj,nl
      real x(nx,ny),y(nx,ny)
      real u(nx,ny,nl),v(nx,ny,nl),den(nx,ny,nl),phi(nx,ny,nl)
      real fsmach,alpha,re,time

      call copen(14,'x2dxray.bin','r',istat)
      call copen(15,'u2dvapr.bin','r',istat)
      call copen(16,'u2dfuel.bin','r',istat)
      call copen(17,'u2dprod.bin','r',istat)

      call cread(15,4,ni,istat)
      call cread(15,4,nj,istat)
      print *,'ni =',ni
      print *,'nj =',nj
      if (ni.ne.nx.or.nj.ne.ny) then
         print *,'wrong grid size !'
         stop
      endif
      call cread(15,4,fsmach,istat)
      call cread(15,4,alpha,istat)
      call cread(15,4,re,istat)
      call cread(15,4,time,istat)
      do 10 j = 1,ny
        do 10 i = 1,nx
          call cread(15,4,den(i,j,1),istat)
 10   continue
      do 20 j = 1,ny
        do 20 i = 1,nx
          call cread(15,4,u(i,j,1),istat)
 20   continue
      do 30 j = 1,ny
        do 30 i = 1,nx
          call cread(15,4,v(i,j,1),istat)
 30   continue
      do 40 j = 1,ny
        do 40 i = 1,nx
          call cread(15,4,phi(i,j,1),istat)
 40   continue
```

```
      call cread(16,4,ni,istat)
      call cread(16,4,nj,istat)
      print *,'ni =',ni
      print *,'nj =',nj
      if (ni.ne.nx.or.nj.ne.ny) then
         print *,'wrong grid size !'
         stop
      endif
      call cread(16,4,fsmach,istat)
      call cread(16,4,alpha,istat)
      call cread(16,4,re,istat)
      call cread(16,4,time,istat)
      do 110 j = 1,ny
        do 110 i = 1,nx
          call cread(16,4,den(i,j,2),istat)
 110   continue
      do 120 j = 1,ny
        do 120 i = 1,nx
          call cread(16,4,u(i,j,2),istat)
 120   continue
      do 130 j = 1,ny
        do 130 i = 1,nx
          call cread(16,4,v(i,j,2),istat)
 130   continue
      do 140 j = 1,ny
        do 140 i = 1,nx
          call cread(16,4,phi(i,j,2),istat)
 140   continue

      call cread(17,4,ni,istat)
      call cread(17,4,nj,istat)
      print *,'ni =',ni
      print *,'nj =',nj
      if (ni.ne.nx.or.nj.ne.ny) then
         print *,'wrong grid size !'
         stop
      endif
      call cread(17,4,fsmach,istat)
      call cread(17,4,alpha,istat)
      call cread(17,4,re,istat)
      call cread(17,4,time,istat)
      do 210 j = 1,ny
        do 210 i = 1,nx
          call cread(17,4,den(i,j,3),istat)
 210   continue
      do 220 j = 1,ny
        do 220 i = 1,nx
          call cread(17,4,u(i,j,3),istat)
 220   continue
      do 230 j = 1,ny
        do 230 i = 1,nx
          call cread(17,4,v(i,j,3),istat)
 230   continue
      do 240 j = 1,ny
        do 240 i = 1,nx
```

```
          call cread(17,4,phi(i,j,3),istat)
 240   continue

      call cread(14,4,ni,istat)
      call cread(14,4,nj,istat)
      print *,'ni =',ni
      print *,'nj =',nj
      if (ni.ne.nx.or.nj.ne.ny) then
         print *,'wrong grid size !'
         stop
      endif
      do 250 j = 1,ny
        do 250 i = 1,nx
          call cread(14,4,x(i,j),istat)
 250   continue
      do 260 j = 1,ny
        do 260 i = 1,nx
          call cread(14,4,y(i,j),istat)
 260   continue

      return
      end
*
*
*
      subroutine igraph(nobj)

* initialize graphics window and draw boundries

#     include "fgl.h"
#     include "fdevice.h"

      real xmax,xmin,ymax,ymin,xbnd,ybnd,xup,xlow,yup,ylow
      real cvect(4),vect(3)
      common/bnd/xmax,xmin,ymax,ymin

      dx = (xmax-xmin)
      dy = (ymax-ymin)
      xup = xmax + .1*dx
      xlow = xmin - .1*dx
      yup = ymax + .3*dy
      ylow = ymin -dy -.3*dy
      zlow = -1.0
      zhi = 1.0
      call foregr
      call prefpo(0,640,0,512)
      iwop = winope('Particle Trace',14)
      call winpop
      call concav(.true.)
      call double
      call RGBmod
      call gconfi
      call ortho(xlow,xup,ylow,yup,zlow,zhi)
      call zbuffe(TRUE)
      call zclear
      return
      end
*
*
*
      subroutine headng(nx,ny,dt)

* display heading and message to start new particle trace

#     include "fgl.h"
#     include "fdevice.h"

      integer nx,ny,sx,sy
      integer lft,rght,bttm,tp
      real xmax,xmin,ymax,ymin,dt,cvect(4),vert(2)
      common/bnd/xmax,xmin,ymax,ymin

      dx = (xmax-xmin)
      dy = (ymax-ymin)
      xup = xmax + .1*dx
      xlow = xmin - .1*dx
      yup = ymax + .3*dy
      ylow = ymin -dy -.3*dy

      call frontb(.true.)
      cvect(1) = 0.5
      cvect(2) = 0.5
      cvect(3) = 0.5
      cvect(4) = 1.0
      call c4f(cvect)
      xp = xmin
      yp = ymin - .2*dy
      call cmov2(xp,yp)
      call charst('Animated particle trace #10                   ',46)
      yp = yp - (32./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('  CFD data files: x2dxray.bin, x2dfuel.bin    ',46)
      yp = yp - (15./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('                  x2dprod.bin, x2dvapr.bin    ',46)
      yp = yp - (24./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('   grid size: ( 29 , 23 )                     ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('   (xmin,xmax): ( 0.0 , 1.049 ) m             ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('   (ymin,ymax): ( 0.0 , .3937 ) m             ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('   dt: 2.342e-2 m/s                           ',46)
      yp = yp - (24./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('Separate particle traces for each phase        ',46)
      yp = yp - (15./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('and phases distinguished through color         ',46)
      yp = yp - (20./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('Color intensity of phases controlled           ',46)
      yp = yp - (15./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('through volume fraction data                   ',46)
      call bgnlin
        vert(1) = xmax - .05*dx
        vert(2) = ymin - .2*dy
        call v2f(vert)
        vert(1) = xmax
        vert(2) = ymin - .2*dy - .05*dx
        call v2f(vert)
        vert(1) = xmax
        vert(2) = ymin - .2*dy - .03*dx
        call v2f(vert)
      call endlin
      call bgnlin
        vert(1) = xmax
        vert(2) = ymin - .2*dy - .05*dx
        call v2f(vert)
        vert(1) = xmax - .02*dx
        vert(2) = ymin - .2*dy - .05*dx
        call v2f(vert)
      call endlin
      xp = xmax
      yp = ymin - .2*dy
      call cmov2(xp,yp)
      call charst('g',1)

      cvect(1) = 0.0
      cvect(2) = 1.0
      cvect(3) = 0.0
      cvect(4) = 1.0
      call c4f(cvect)
      xp = xmax - .25*dx
      yp = ymin - .2*dy - .25*dx
      call cmov2(xp,yp)
      call charst('GREEN: Fuel',11)
      cvect(1) = 0.0
      cvect(2) = 0.0
      cvect(3) = 1.0
      cvect(4) = 1.0
      call c4f(cvect)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('BLUE: Product',13)
      cvect(1) = 1.0
      cvect(2) = 0.0
      cvect(3) = 0.0
      cvect(4) = 1.0
      call c4f(cvect)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('RED: vapor',10)

      call frontb(.false.)

      return
      end
*
*
*
      subroutine defobj(nobj,nx,ny,nl,x,y,u,v)

* define graphical objects #1 to #nobj for the paths,
* GRID for the grid, and REACT for the reactor boundries
* and BACKGR for the black background to erase the particles

#     include "fgl.h"
#     include "fdevice.h"

      integer nobj,GRID,REACT,nx,ny
      real x(nx,ny),y(nx,ny),u(nx,ny,nl),v(nx,ny,nl)
      real vert(3),cvect(4),vect(3)
      real xmax,xmin,ymax,ymin
      common/bnd/xmax,xmin,ymax,ymin

      GRID = nobj+1
      REACT = nobj+2
      BACKGR = nobj+3

      do 150 j = 1,nobj
        call makeob(j)
          call linewi(3)
        call closeo(j)
 150  continue

      call makeob(GRID)
        cvect(1) = .04
        cvect(2) = .04
        cvect(3) = .04
        cvect(4) = 1.
        call c4f(cvect)
        call linewi(1)
        do 250 i = 1,nx
          call bgnlin
            do 200 j = 1,ny
              vert(1) = x(i,j)
              vert(2) = y(i,j)
              vert(3) = 1.0
              call v3f(vert)
200         continue
          call endlin
250     continue
        do 350 j = 1,ny
```

```
          call bgnlin
            do 300 i = 1,nx
              vert(1) = x(i,j)
              vert(2) = y(i,j)
              vert(3) = 1.0
              call v3f(vert)
300         continue
          call endlin
350     continue
      call closeo(GRID)

      call makeob(REACT)
        cvect(1) = 0.
        cvect(2) = 0.
        cvect(3) = 0.5
        cvect(4) = 1.
        call c4f(cvect)
        call linewi(3)
        do 400 j = 1,ny-1
         do 400 i = 1,nx-1
           flag1 = 1.
           flag2 = 1.
           flag3 = 1.
           flag4 = 1.
           flagt = 1.
           do 450 l = 1,nl
             if (u(i,j,l).ne.0.0.or.v(i,j,l).ne.0.0) then
               flag1 = 0
             endif
             if (u(i+1,j,l).ne.0.0.or.v(i+1,j,l).ne.0.0) then
               flag2 = 0
             endif
             if (u(i,j+1,l).ne.0.0.or.v(i,j+1,l).ne.0.0) then
               flag3 = 0
             endif
             if (u(i+1,j+1,l).ne.0.0.or.v(i+1,j+1,l).ne.0.0) then
               flag4 = 0
             endif
 450       continue
           flagt = flag1+flag2+flag3+flag4
           if (flagt.ge.2) then
*
*     draw solid surface
*
             if (flagt.eq.4) then
               call bgnpol
                 vert(1) = x(i,j)
                 vert(2) = y(i,j)
                 vert(3) = 1.0
                 call v3f(vert)
                 vert(1) = x(i+1,j)
                 vert(2) = y(i+1,j)
                 vert(3) = 1.0
                 call v3f(vert)
                 vert(1) = x(i+1,j+1)
              vert(2) = y(i+1,j+1)
              vert(3) = 1.0
              call v3f(vert)
              vert(1) = x(i,j+1)
              vert(2) = y(i,j+1)
              vert(3) = 1.0
              call v3f(vert)
            call endpol
          endif
*
*     draw solid walls
*
          if (flag1.eq.1.and.flag2.eq.1) then
            call bgnlin
              vert(1) = x(i,j)
              vert(2) = y(i,j)
              vert(3) = 1.0
              call v3f(vert)
              vert(1) = x(i+1,j)
              vert(2) = y(i+1,j)
              vert(3) = 1.0
              call v3f(vert)
            call endlin
          endif
          if (flag1.eq.1.and.flag3.eq.1) then
            call bgnlin
              vert(1) = x(i,j)
              vert(2) = y(i,j)
              vert(3) = 1.0
              call v3f(vert)
              vert(1) = x(i,j+1)
              vert(2) = y(i,j+1)
              vert(3) = 1.0
              call v3f(vert)
             call endlin
           endif
           if (flag4.eq.1.and.flag2.eq.1) then
             call bgnlin
               vert(1) = x(i+1,j+1)
               vert(2) = y(i+1,j+1)
               vert(3) = 1.0
               call v3f(vert)
               vert(1) = x(i+1,j)
               vert(2) = y(i+1,j)
               vert(3) = 1.0
               call v3f(vert)
             call endlin
           endif
           if (flag4.eq.1.and.flag3.eq.1) then
             call bgnlin
               vert(1) = x(i+1,j+1)
               vert(2) = y(i+1,j+1)
               vert(3) = 1.0
               call v3f(vert)
               vert(1) = x(i,j+1)
```

```
                vert(2) = y(i,j+1)
                vert(3) = 1.0
                call v3f(vert)
              call endlin
            endif
          endif
 400    continue
      call closeo(REACT)

* define graphical object for the BLACK background to erase the particles

      dx = (xmax-xmin)
      dy = (ymax-ymin)
      xup = xmax + .1*dx
      xlow = xmin - .1*dx
      yup = ymax + .3*dy
      ylow = ymin -dy -.3*dy
      cvect(1) = 0.
      cvect(2) = 0.
      cvect(3) = 0.
      cvect(4) = 1.
      vect(1) = xmin
      vect(2) = ymin
      vect(3) = 0.0
      call makeob(BACKGR)
        call bgnpol
          call c4f(cvect)
          call v3f(vect)
      call closeo(BACKGR)
      vect(1) = xmax
      vect(2) = ymin
      vect(3) = 0.0
      call editob(BACKGR)
          call v3f(vect)
      call closeo(BACKGR)
      vect(1) = xmax
      vect(2) = ymax
      vect(3) = 0.0
      call editob(BACKGR)
          call v3f(vect)
      call closeo(BACKGR)
      vect(1) = xmin
      vect(2) = ymax
      vect(3) = 0.0
      call editob(BACKGR)
          call v3f(vect)
        call endpol
      call closeo(BACKGR)

      return
      end
*
*
*
      subroutine dinit(nobj)
```

```
* initialize graphics drawing

#     include "fgl.h"
#     include "fdevice.h"

      integer nobj,GRID,REACT
      real xmax,xmin,ymax,ymin,cvect(4)
      common/bnd/xmax,xmin,ymax,ymin

      GRID = nobj+1
      REACT = nobj+2
      cvect(1) = 0.
      cvect(2) = 0.
      cvect(3) = 0.
      cvect(4) = 1.
      call c4f(cvect)
      call frontb(.true.)
      call clear
      call callob(GRID)
      call callob(REACT)
      call frontb(.false.)

      return
      end
*
*
*
      subroutine modobj(nx,ny,nobj,l,x,y,px,py,pphi,maxp,np)

* modify the graphical objects by adding the path recently defined

#     include "fgl.h"
#     include "fdevice.h"

      integer nx,ny,nl,nobj,maxp,np,bsplin
      real x(nx,ny),y(nx,ny),pphi(maxp),cvect(4)
      real px(maxp),py(maxp),xpt,ypt,geom(3,4)
      real xmax,xmin,ymax,ymin,rad,msix,asix,tthr,bspmat(4,4)
      common/bnd/xmax,xmin,ymax,ymin
      parameter (bsplin=3,msix=-1.0/6.0,asix=1.0/6.0,tthr=2.0/3.0)

      data bspmat/ msix,   .5,  -.5,  asix,
     +                 .5, -1.0,   .5,   0.0,
     +                -.5,  0.0,   .5,   0.0,
     +              asix, tthr, asix,   0.0/

      cvect(1) = 0.
      cvect(2) = 0.
      cvect(3) = 0.
      cvect(4) = 0.
      cvect(l) = 1.0
      npm3 = np-3

* represent particles with B-splines through four consecutive time steps
* for a particle trace, draw a particle every nobj timestep, and
* increment one timestep each graphical object

* define an alpha value (cvect(4)) to control transparency through
* volume fraction data (pphi)

      call defbas(bsplin,bspmat)
      call curveb(bsplin)
      do 100 i = 1,nobj
        do 200 j = i,npm3,nobj
          ic = 0
          phisum = 0.0
          do 300 k = j,j+3
            ic = ic+1
            phisum = phisum + pphi(k)
            geom(1,ic) = px(k)
            geom(2,ic) = py(k)
            geom(3,ic) = 0.0
 300      continue

* define an alpha value (cvect(4)) to control transparency through
* volume fraction data (pphi)

          phiave = phisum/4.0
          cvect(4) = amax1(0.0,phiave)
          call editob(i)
            call c4f(cvect)
            call crv(geom)
          call closeo(i)
 200    continue

* eliminate disappearing particles to avoid blinking during animation

        if (i.gt.npm3) then
          j = i-((i-1)/npm3)*npm3
          ic = 0
          phisum = 0.0
          do 400 k = j,j+3
            ic = ic+1
            phisum = phisum + pphi(k)
            geom(1,ic) = px(k)
            geom(2,ic) = py(k)
            geom(3,ic) = 0.0
 400      continue
          phiave = phisum/4.0
          cvect(4) = amax1(0.0,phiave)
          call editob(i)
            call c4f(cvect)
            call crv(geom)
          call closeo(i)
        endif
 100  continue

      return
      end
*
*
*
      subroutine transp(nx,ny,maxp,np,x,y,phil,px,py,pphi)

* calculate particle transparency based on volume fraction data

      integer nx,ny,maxp,np,md,od
      real x(nx,ny),y(nx,ny),phil(nx,ny)
      real px(maxp),py(maxp),pphi(maxp)
      real xc,yc,xx,yy,b,a,c1,c2,e(4)

      do 100 ii = 1,np
        od = 0
        md = 0
        jj = ii + 1
        call ploc(nx,ny,x,y,px,py,maxp,jj,od,md)
        if (od.ne.0) then
          i = (od-1)/ny + 1
          j = od - (i-1)*ny
          pphi(ii) = phil(i,j)
        else if (md.ne.0) then
          i = (md-1)/ny + 1
          j = md - (i-1)*ny
          xc = (x(i+1,j) + x(i,j))/2.0
          yc = (y(i,j+1) + y(i,j))/2.0
          b = (x(i+1,j) - x(i,j))/2.0
          a = (y(i,j+1) - y(i,j))/2.0
          xx = px(ii) - xc
          yy = py(ii) - yc
          c1 = xx/b
          c2 = yy/a
          e(1) = .25*(1-c1)*(1-c2)
          e(2) = .25*(1+c1)*(1-c2)
          e(3) = .25*(1+c1)*(1+c2)
          e(4) = .25*(1-c1)*(1+c2)
          pphi(ii)=e(1)*phil(i,j)+e(2)*phil(i+1,j)+
     &              e(3)*phil(i+1,j+1)+e(4)*phil(i,j+1)
        else
          print *,'particle out of grid'
          stop
        endif
 100  continue

      return
      end
*
*
*
```

```
      integer nx,ny,np,maxp,nobj,iflag
      parameter (nx=46,ny=24,maxp=500,nobj=497)
      real x(nx,ny),y(nx,ny),u(nx,ny),v(nx,ny)
      real px(maxp),py(maxp),dt,xint,yint
      real xmax,xmin,ymax,ymin
      common/bnd/xmax,xmin,ymax,ymin

      call getdat(nx,ny,x,y,u,v)
      call edges(nx,ny,x,y)
      call dtime(nx,ny,x,y,u,v,dt)
      call igraph
      call defobj(nobj,nx,ny,x,y,u,v)
      call dinit(nobj)
      call headng(nx,ny,dt)

* compute 98 particle traces for the timeline

      xint = xmin + (xmax-xmin)/2.
      dy = (ymax-ymin)/100
      do 50 npt = 1,98
        yint = ymin + npt*dy
        call trace(nx,ny,x,y,u,v,xint,yint,px,py,maxp,np,dt,nflag)
        if (nflag.eq.0) then
          print *,'not added to graphical objects'
        else
          call modobj(nobj,px,py,maxp,np)
        endif
 50   continue

* begin animation loop

      call writem(7)
      do while (1)
        call color(BLACK)
        call clear
        call callob(1)
        call swapbu
        call sleep(5)
        do 100 i = 2,nobj
          call color(BLACK)
          call clear
          call callob(i)
          call swapbu
 100    continue
      end do
      call qreset

      end
*
*
*
      subroutine headng(nx,ny,dt)

* display heading and message to start new particle trace
```

```
#     include "fgl.h"
#     include "fdevice.h"

      integer nx,ny,sx,sy
      integer lft,rght,bttm,tp
      real xmax,xmin,ymax,ymin,dt
      common/bnd/xmax,xmin,ymax,ymin

      do 100 k = 1,8
        call mapcol(655+k,100,100,100)
 100  continue
      dx = (xmax-xmin)
      dy = (ymax-ymin)
      xup = xmax + .1*dx
      xlow = xmin - .1*dx
      yup = ymax + .3*dy
      ylow = ymin -dy -.3*dy

      call frontb(.true.)
      call color(656)
      xp = xmin
      yp = ymin - .2*dy
      call cmov2(xp,yp)
      call charst('Animated particle trace #3                    ',46)
      yp = yp - (32./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('  CFD data file: mint.bin                     ',46)
      yp = yp - (24./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('   grid size: ( 46 , 24 )                     ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('   (xmin,xmax): ( 0.0 , .3556 ) m             ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('   (ymin,ymax): ( 0.0 , .04445 ) m            ',46)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('   dt: .007347 m/s                            ',46)
      yp = yp - (24./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('Initial column of particle traces calculated to',47)
      yp = yp - (18./512.)*(yup - ylow)
      call cmov2(xp,yp)
      call charst('generate a timeline. Total # particle traces: 98',48)
      call frontb(.false.)
      return
      end
*
*
*
```

References

1. White, Frank M., Fluid Mechanics, 2nd ed., McGraw Hill Book Co., New York,1986.

2. 'Flow Visualization', ASME Film Catalog, Journal of Fluids Engineering, 1976

3. 'Computer Generated Flow Visualizatiom Motion Pictures', NASA Lewis Research Center, Cleveland OH

4. Van Dyke, Milton, An Album of Fluid Motion, Parabolic Press, Stanford, CA, 1982

5. Merzkirich, W., Flow Visualization, Academic Press, New York, 1974.

6. International Symposium on Physical and Numerical Flow Visualization, Joint ASCE/ASME Mechanics Conference, Albequerque, NM, 1985.

7. Asanuma, T., Flow Visualization, International Symposium on Flow Visualization, Tokoyo, Japan, 1977.

8. Yang, Wen-Jei, Handbook of Flow Visualization, Hemisphere Pub. Co., New York, 1989.

9. Flow Visualization -1989, Winter Anula Meeting of the ASME, San Francisco, CA, 1989.

10. Flow Visualization III, 3rd International Symposium on Flow Visualization, Ann Arbor, MI, 1985.

11. Flow Visualization IV, 4th International Symposium on Flow Visualization, Paris, France, 1986.

12. Parnell, L.A. et. al., AIAA 87-1808, AIAA/ASME/SAE/ASEE 23rd Joint Propulsion Conference, Montery, CA, 1989.

13. Parnell, L.A. et. al., AIAA 89-2825, AIAA/ASME/SAE/ASEE 25th Joint Propulsion Conference, Montery, CA, 1989.

14. Buning, P., Walatka, P.P., **PLOT 3D,** Sterling Software, NASA Ames Research Center, 1988

15. Walatka, P.P., **FAST**, NASA Ames Resaerch Center WAO RND, 1991

16. Buning, P.G. et. al., AIAA 85-1507, AIAA 7th Computational Fluid Dyanamics Conference, Cincinatti, OH 1985.

17. Buning, P.G. et. al., 'Flow Visualization of CFD using Graphics Workstations', AIAA 8th Computational Fluid Dyanamics Conference, Honolulu,1987.

18. Buning, P.G. et. al., 'Use of Computer Graphics for Visualization of Flow Fields', AIAA Aerospace Engineering Conference, Los Angeles, 1987.

19. Miller, T.F., 'Numerical Simulation of the Flowfield in a Liquid Metal Combustion Chamber', 23rd JANNAF Combustion Meeting,. Cheyene, WY, 1990

20. Globus, A., Levit, C., and Lasinski, T., 'A Tool for Visualizing the Topology of Three-Dimensional Vector Fields', Report RNR-91-017, Moffet Field, CA, 1991.

21. Helman, J.L., and Hesselink, L., 'Visualizing Vector Field Topology in Fluid Flows', IEEE Computer Graphics and Applications, May, 1991.

22. Thalmann, D., Scientific Visualization and Graphics Simulation, Wiley, Chirchester, 1990.

23. Goss, M.E., 'A Real Time Particle System for Display of Ship Wakes', IEEE Computer Graphics and Applications, May, 1990.

24. Reeves, W.T., P'article Systems- a Technique for Modelling a Class of Fuzzy Objects', Computer Graphics, July, 1983.

25. Ramsden, D. and Holloway, G., 'Timestepping Lagrangian Particles in Two Dimensional Eulerian Flow Fields', Journal of Computational Physics 95, 1991.

26. Segerlind, L.J., Applied Finite Element Analysis, Wiley, New York, 1978.

27. Belegundu, A.D., and Chandrupatla, T.R., Intorduction to Finite Elements in Engineering, Prentice Hall, Englewood, N.J., 1991.